School of Computer Science

CS 343
Concurrent and Parallel Programming

Course Notes* Fall 2015

http://www.student.cs.uwaterloo.ca/~cs343

μC++ download or Github (installation: sudo sh u++-6.1.0.sh)

November 8, 2015

Outline

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

Contents

1 Advanced Control Flow (Review)

- ***Within*** a routine, basic and advanced control structures allow virtually any control flow.
- **Multi-exit loop** (or mid-test loop) has one or more exit locations occurring *within* the body of the loop, not just top (**while**) or bottom (**do-while**):

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

  condition reversed from **while**, outdent exit for readability

- Eliminates priming (duplicated) code necessary with **while**:

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

```
for ( ;; ) {
    cin >> d;
  if ( cin.fail() ) break;
    . . .
}
```

- Eliminate **else** on loop exits:

BAD	GOOD	BAD	GOOD

`for ( ;; ) {`
`    S1`
`    if ( C1 ) {`
`        S2`
`    } else {`
`        break;`
`    }`
`    S3`
`}`
	`for ( ;; ) {`
	`    S1`
	`  if ( ! C1 ) break;`
	`    S2`
	
	`    S3`
	`}`
		`for ( ;; ) {`
		`    S1`
		`    if ( C1 ) {`
		`        break;`
		`    } else {`
		`        S2`
		`    }`
		`    S3`
		`}`
			`for ( ;; ) {`
			`    S1`
			`  if ( C1 ) break;`
			
			`    S2`
			
			`    S3`
			`}`

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

- Allow multiple exit conditions:

```
for ( ;; ) {
    S1
  if ( i >= 10 ) { E1; break; }

    S2
  if ( j >= 10 ) { E2; break; }

    S3
}
```

```
bool flag1 = false, flag2 = false;
while ( ! flag1 & ! flag2 ) {
    S1
    if ( C1 ) flag1 = true;
    } else {
        S2
        if ( C2 ) flag2 = true;
        } else {
            S3
        }
    }
}
if ( flag1 ) E1;
else E2;
```

- Eliminate flag variables necessary with **while**.
  - **flag variable** is used solely to affect control flow, i.e., does not contain data associated with a computation.
- ***Flag variables are the variable equivalent to a goto*** because they can be set/reset/tested at arbitrary locations in a program.
- **Static multi-level exit** exits multiple control structures where exit points are *known* at compile time.
- Labelled exit (**break**/**continue**) provides this capability:

μC++ / Java

```
L1: { // good eye-candy
    ... declarations ...
    L2: switch ( ... ) {
        L3: for ( ... ) {
            ... break L1; ... // exit block
            ... break L2; ... // exit switch
            ... break L3; ... // exit loop
        }
        ...
    }
    ...
}
```

C / C++

```
{
    ... declarations ...
    switch ( ... ) {
        for ( ... ) {
            ... goto L1; ...
            ... goto L2; ...
            ... goto L3; ... // or break
        } L3: ;
        ...
    } L2: ; // bad eye-candy
    ...
} L1: ;
```

- Why is it good practice to label all exits?
- Eliminate all flag variables with multi-level exit!

```
B1: for ( i = 0; i < 10; i += 1 ) {

    B2: for ( j = 0; j < 10; j += 1 ) {

            ...
        if ( ... ) break B2;  // outdent

            ... // rest of loop
    if ( ... ) break B1;  // outdent

            ... // rest of loop


    } // for

    ... // rest of loop

} // for
```

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

- Other uses of multi-level exit to remove duplicate code:

duplication

```
if ( C1 ) {
    S1;
    if ( C2 ) {
        S2;
        if ( C3 ) {
            S3;
        } else
            S4;
    } else
        S4;
} else
    S4;
```

no duplication

```
C: {
    if ( C1 ) {
        S1;
        if ( C2 ) {
            S2;
            if ( C3 ) {
                S3;
                break C;
            }
        }
    }
    S4; // only once
}
```

```
    if ( C1 ) {
        S1;
        if ( C2 ) {
            S2;
            if ( C3 ) {
                S3;
                goto C;
            }
        }
    }
    S4; // only once
C: ;
```

- Normal and labelled **break** are a **goto** with restrictions:

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

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

1.1 Dynamic Memory Allocation

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

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

}
```

```
{ // BAD, unnecessary dynamic allocation
    cin >> size;
    int *arr = new int[size];
    . . . // use arr[i]
    delete [] arr; // why "[ ]"?
}
```

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

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

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

  2. When the amount of data read is unknown.

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

Dynamic allocation is implicit.

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

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

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

Alternatives are static variables or initialization after creation ($\Rightarrow$ no constructor).

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

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

```
_Coroutine C {
    void main() {
        int *arr = new int[100000];
        ...
    }
};
```

Alternatives are large stacks (waste virtual space) or dynamic stack growth (complex and pauses).

2 Exceptions

2.1 Dynamic Multi-Level Exit

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

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

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

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

- **Fails to compile because labels only have routine scope.**

- ⇒ ***among*** routines, control flow is controlled by call/return mechanism.

    - given A calls B calls C, it is impossible to transfer directly from C back to A, terminating B in the transfer.

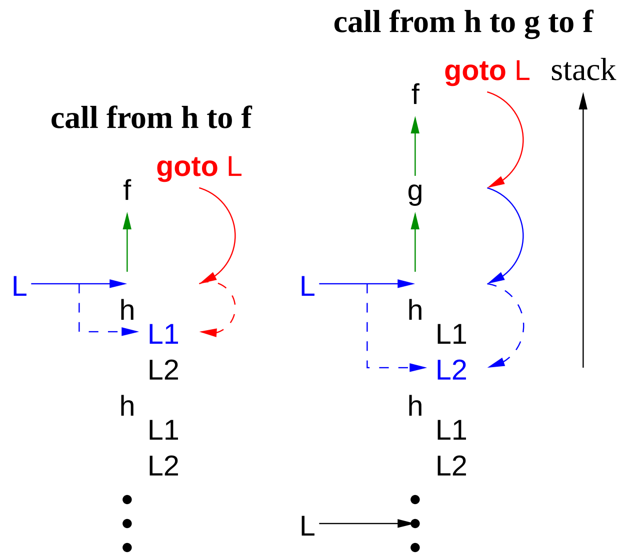
- Fundamentally, a routine can have multiple kinds of return.

    - routine call returns normally, i.e., statement after the call
    - exceptional returns, i.e., control transfers to statements **not** after the call

```
C     Two alternate return parameters, denoted by * and implicitly named 1 and 2
         subroutine AltRet( c, *, * )
            integer c;
            if ( c == 0 ) return       ! normal return
            if ( c == 1 ) return 1     ! alternate return
            if ( c == 2 ) return 2     ! alternate return
         end
C     Statements labelled 10 and 20 are alternate return points
         call AltRet( 0, *10, *20 )
         print *, "normal return 1"
         call AltRet( 1, *10, *20 )
         print *, "normal return 2"
         return
10       print *, "alternate return 1"
         call AltRet( 2, *10, *20 )
         print *, "normal return 3"
         return
20       print *, "alternate return 2"
         stop
         end
```

- Generalization of multi-exit loop and multi-level exit.
  - control structures end with or without an exceptional transfer.
- Pattern addresses fact that:
  - algorithms can have multiple outcomes
  - separating outcomes makes it easy to read and maintain a program
- Pattern does not handle case of multiple levels of nested modularization.
  - if AltRet is further modularized, new routine has to have an alternate return to AltRet and then another alternate return to its caller.
  - Rather than two step operation, simpler for new modularized routine to bypass intermediate step and transfer directly to caller of AltRet.
- **Dynamic multi-level exit** extend call/return semantics to transfer in the ***reverse*** direction to normal routine calls, called **non-local transfer**.

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



- Non-local transfer mechanism is a label variable containing the tuple:
  1. pointer to a block activation on the stack
  2. transfer point within the block.
- Non-local transfer, **goto** L, in f is a two-step operation:
  1. direct control flow to the specified activation on the stack;
  2. then go to the transfer point (label) within the routine.
- Therefore, a label value is not statically/lexically determined.
  - recursion in g $\Rightarrow$ unknown distance between f and h on stack.
  - what if L is set during the recursion of h?
- ***Transfer between* goto *and label value causes termination of stack block.***
- First non-local transfer from f transfers to the label L1 in h's routine activation, terminating f's activation.
- Second non-local transfer from f transfers to the static label L2 in the stack frame for h, terminating the stack frame for f and g.
- Termination is implicit for direct transferring to h or requires stack unwinding if activations contain objects with destructors or finalizers.
- Non-local transfer is possible in C using:
  - jmp_buf to declare a label variable,

   - setjmp to initialize a label variable,
   - longjmp to goto a label variable.

- Non-local transfer allows multiple forms of returns to any level.
   - Normal return transfers to statement after the call, often implying completion of routine's algorithm.
   - Exceptional return transfers to statement **not** after the call, indicating an ancillary completion (but not necessarily an error).

- Unfortunately, non-local transfer is too general, allowing branching to almost anywhere, i.e., the goto problem.

2.2 Traditional Approaches

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

  e.g., printf() returns number of bytes transmitted or negative value.

- **status flag**: set shared (global) variable indicating normal or exceptional execution; the value remains as long as it is not overwritten.

  e.g., errno variable in UNIX.

- **fix-up routine**: a global and/or local routine called for an exceptional event to fix-up and return a corrective result so a computation can continue.

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

  e.g., C++ has global routine-pointer new_handler called when **new** fails.

- Techniques are often combined, e.g.:

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

- Drawbacks of traditional techniques:
   - checking return code or status flag is optional ⇒ can be delayed or omitted, i.e., passive versus active
   - return code mixes exceptional and normal values ⇒ enlarges range of values for computation; normal/exceptional values should be independent
   - testing and handling of return code or status flag is often done locally (inline), otherwise information may be lost; but local testing/handling:
      * makes code difficult to read; each call results in multiple statements
      * can be inappropriate; library routines should not terminate program

- ○ local fix-up routines increases the number of parameters
  - ∗ increase cost of each call
  - ∗ must be passed through multiple levels enlarging parameter lists even when the fix-up routine is not used
- ○ non-local (non-inline) return code or status flag testing is difficult because multiple values must be returned to higher-level code for subsequent analysis, compounding the mixing problem
- ○ status flag can be overwritten before examined, and cannot be used in a concurrent environment because of sharing issues
- ○ non-local (global) fix-up routines, often implemented using a global routine-pointer, have identical problems with status flags.

- Simulate dynamic multi-level exit with return codes.

```
label L;
void f( int i, int j ) {

    for ( . . . ) {
        int k;
        . . .
  if ( i < j && k > i ) goto L;
        . . .
    }


}
void g( int i ) {

    for ( . . . ) {
        int j;
        . . . f( i, j ); . . .

    }


}
void h() {
    L = L1;
    for ( . . . ) {
        int i;
        . . . g( i ); . . .
    }
    L1: . . .
}
```

```
int f( int i, int j ) {
    bool flag = false;
    for ( ! flag && . . . ) {
        int k;
        . . .
  if ( i < j && k > i ) flag = true;
        else { . . . }
    }
    if ( ! flag ) { . . . }
    return flag ? -1 : 0;
}
int g( int i ) {
    bool flag = false;
    for ( ! flag && . . . ) {
        int j;
        if ( f( i, j ) == -1 ) flag = true
        else { . . . }
    }
    if ( ! flag ) { . . . }
    return flag ? -1 : 0;
}
void h() {
    bool flag = false;
    for ( ! flag && . . . ) {
        int i;
        if ( g( i ) == -1 ) flag = true;
        else { . . . }
    }
}
```

2.3 Exception Handling

- Dynamic multi-level exit allows complex forms of transfers among routines.

- Complex control-flow among routines is often called **exception handling**.
- Exception handling is more than error handling.
- An **exceptional event** is an event that is (usually) known to exist but which is *ancillary* to an algorithm.
    - an exceptional event usually occurs with low frequency
    - e.g., division by zero, I/O failure, end of file, pop empty stack
- An **exception handling mechanism** (EHM) provides some or all of the alternate kinds of control-flow.
- Very difficult to simulate EHM with simpler control structures.
- Exceptions are supposed to make certain programming tasks easier, like robust programs.
- Robustness results because exceptions are active versus passive, forcing programs to react immediately when an exceptional event occurs.
- An EHM is not a panacea and only as good as the programmer using it.

2.4 Execution Environment

- The execution environment has a significant effect on an EHM.
- An object-oriented concurrent environment requires a more complex EHM than a non-object-oriented sequential environment.
- E.g., objects may have destructors that must be executed no matter how the object ends, i.e., by normal or exceptional termination.

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

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

Java	μC++

<pre>try {
    infile = new Scanner(new File("abc"));
    ... if (...) throw new E();
    ...
} finally { // always executed
    infile.close();   // must close file
}</pre>
	<pre>try {
	    infile = new ifstream("abc");
	    ... if (...) throw new E();
	    ...
	} _Finally { // always executed
	    infile.close(); // must close file
	}</pre>

- Hence, terminating a block complicates the EHM as object destructors (and recursively for nested objects) and **finally** clauses must be executed.
- Another example is complex execution-environment involving continuation, coroutine, task, each with its own execution stack.
- Given multiple stacks, an EHM can be more sophisticated, resulting in more complexity.
  - e.g., if no handler is found in one stack, it is possible to continue propagating the exception in another stack.

2.5 Terminology

- **execution** is the language unit in which an exception can be raised, usually any entity with its own runtime stack.
- **exception type** is a type name representing an exceptional event.
- **exception** is an instance of an exception type, generated by executing an operation indicating an ancillary (exceptional) situation in execution.
- **raise** (**throw**) is the special operation that creates an exception.
- **source execution** is the execution raising an exception.
- **faulting execution** is the execution changing control flow due to a raised exception.
- **local exception** is when an exception is raised and handled by the same execution $\Rightarrow$ source = faulting.
- **non-local exception** is when an exception is raised by a source execution but **delivered** to a different faulting execution $\Rightarrow$ source $\neq$ faulting.
- **concurrent exception** is a non-local exception, where the source and faulting executions are executing concurrently.
- **propagation** directs control from a raise in the source execution to a handler in the faulting execution.
- **propagation mechanism** is the rules used to locate a handler.
  - most common propagation-mechanisms give precedence to handlers higher in the lexical/call stack
    - specificity versus generality
    - efficient linear search during propagation
- **handler** is inline (nested) routine responsible for handling raised exception.
  - handler **catch**es exception by **match**ing with one or more exception types

- ○ after catching, a handler executes like a normal subroutine
- ○ handler can return, reraise the current exception, or raise a new exception
- ○ **reraise** terminate current handling and continuing propagation of caught exception.
  - ∗ useful if a handler cannot deal with an exception but needs to propagate same exception to handler further down the stack.
  - ∗ provided by a raise statement without an exception type:

    ```
    ... throw;    // no exception type
    ```

    where a raise must be in progress.
- ○ an exception is **handled** only if the handler returns rather than reraises

- **guarded block** is a language block with associated handlers, e.g., try-block in C++/Java.
- **unguarded block** is a block with no handlers.
- **termination** means control cannot return to the raise point.
  - ○ all blocks on the faulting stack from the raise block to the guarded block handling the exception are terminated, called **stack unwinding**
- **resumption** means control returns to the raise point ⇒ no stack unwinding.
- EHM = Exception Type + Raise (exception) + Propagation + Handlers

2.6 Static/Dynamic Call/Return

- All routine/exceptional control-flow can be characterized by two properties:
  1. static/dynamic call: routine/exception name at the call/raise is looked up statically (compile-time) or dynamically (runtime).
  2. static/dynamic return: after a routine/handler completes, it returns to its static (definition) or dynamic (call) context.

	call/raise	
return/handled	static	dynamic
static	1) sequel	3) termination exception
dynamic	2) routine	4) routine pointer, virtual routine, resumption

- E.g., case 2) is a normal routine, with static name lookup at the call and a dynamic return.

2.7 Static Propagation (Sequel)

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

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

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

- Without a sequel, it is impossible to modularize code with static exits.
- ⇒ propagation is along the lexical structure
- Adheres to the termination model, as the stack is unwound.
- Sequel handles termination for a *non-recoverable* operation.

```
{ // new block
    sequel StackOverflow(...) { ... } // handler
    class stack {
        void push( int i ) {
            if (...) StackOverflow(...);
        }
        ...
    };
    stack s;
    ... s.push( 3 ); ... // overflow ?
} // sequel returns here
```

- The advantage of the sequel is the handler is statically known (like static multi-level exit), and can be as efficient as a direct transfer.
- The disadvantage is that the sequel only works for monolithic programs because it must be statically nested at the point of use.

- Fails for modular (library) code as the static context of the module and user code are disjoint.
- E.g., if stack is separately compiled, the sequel call in push no longer knows the static blocks containing calls to it.

2.8 Dynamic Propagation

- Cases 3) and 4) are called termination and resumption, and both have dynamic raise with static/dynamic return, respectively.
- Dynamic propagation/static return (case 3) is also called dynamic multi-level exit (see Section 2.1, p. 5).
- The advantage is that dynamic propagation works for separately-compiled programs.
- The disadvantage (advantage) of dynamic propagation is the handler is not statically known.
  - without dynamic handler selection, the same action and context for that action is executed for every exceptional change in control flow.

2.8.1 Termination

- For termination:
  - control transfers from the start of propagation to a handler ⇒ dynamic raise (call)
  - when handler returns, it performs a static return ⇒ stack is unwound (like sequel)
- There are 3 basic termination forms for a *non-recoverable* operation: non-local, terminate, and retry.
- Non-local transfer provides *general* mechanism for block transfer on call stack, but has **goto** problem (see Section 2.1, p. 5).
- **terminate** provides *limited* mechanism for block transfer on the call stack (like labelled **break**):

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

```
label L;
void f(...) {
    ...
    goto L;

}
int main() {
    L = L1; // set transfer-point
    f(...); goto S1;
  L1: // handle non-local return
  S1: L = L2; // set transfer-point
    f(...); goto S2;
  L2: // handle non-local return
  S2: ; ...
}
```

- **retry** is a combination of termination with special handler semantics, i.e., restart the guarded block handling the exception (Eiffel). (Pretend end-of-file is an exception of type Eof.)

Retry

```
char readfiles( char *files[], int N ) {
    int i = 0, value;
    ifstream infile;
    infile.open( files[i] );

    try {
        ... infile >> value; ...
    } retry( Eof ) {
        i += 1;
        infile.close();
      if ( i == N ) goto Finished;
        infile.open( files[i] );
    }
  Finished: ;
}
```

Simulation

```
char readfiles( char *files[], int N ) {
    int i = 0, value;
    ifstream infile;
    infile.open( files[i] );
    while ( true ) {
        try {
            ... infile >> value; ...
        } catch( eof ) {
            i += 1;
            infile.close();
          if ( i == N ) break;
            infile.open( files[i] );
        }
    }
}
```

- Because retry can be simulated, it is seldom supported directly.

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

C++

```
ifstream infile;
ofstream outfile;
outfile.exceptions( ios_base::failbit );
infile.exceptions( ios_base::failbit );
switch ( argc ) {
  case 3:
    try {
        outfile.open( argv[2] );
    } catch( ios_base::failure ) {...}
    // fall through to handle input file
  case 2:
    try {
        infile.open( argv[1] );
    } catch( ios_base::failure ) {...}
    break;
  default:
    ...
} // switch
string line;
try {
    for ( ;; ) {  // loop until end-of-file
        getline( infile, line );
        outfile << line << endl;
    }
} catch ( ios_base::failure ) {}
```

μC++

```
ifstream infile;
ofstream outfile;


switch ( argc ) {
  case 3:
    try {
        outfile.open( argv[2] );
    } catch( uFile::Failure ) {...}
    // fall through to handle input file
  case 2:
    try {
        infile.open( argv[1] );
    } catch( uFile::Failure ) {...}
    break;
  default:
    ...
} // switch
string line;

for ( ;; ) {
    getline( infile, line );
  if ( infile.fail() ) break;
    outfile << line << endl;
}
```

  - ios::exception mask indicates stream state-flags throw an exception if set

- ○ failure exception raised after failed open or end-of-file when failbit set in exception mask
- ○ μC++ provides exceptions for I/O errors, but no exception for eof.

- An exception handler can generated an arbitrary number of nested exceptions.

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

h ~~f~~
f
f
h ~~f~~ throw E_2
f
f
h ~~f~~ throw E_1
f
f

Exceptions are nested as handler can rethrow its matched exception when control returned.

- A destructor ***can*** raise an exception.

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

```
y's destructor
    | throw E
inner try           x's destructor
    | y                 | throw E
outer try           outer try
    | x                 | x
```

- ○ y's destructor called at end of inner **try** block, it raises an exception E, which unwinds destructor and **try**, and handled at inner **catch**
- ○ x's destructor called at end of outer **try** block, it raises an exception E, which unwinds destructor and **try**, and handled at outer **catch**

- A destructor ***cannot*** raise an exception during propagation.

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

1. raise of E causes unwind of inner **try** block
2. x's destructor called during unwind, it raises an exception E, which terminates program
3. Cannot start second exception without handler to deal with first exception, i.e., cannot drop exception and start another.
4. Cannot postpone first exception because second exception may remove its handlers during stack unwinding.

2.8.2 Resumption

- **resumption** provides a *limited* mechanism to generate new blocks on the call stack:
  - control transfers from the start of propagation to a handler ⇒ dynamic raise (call)
  - when handler returns, it is dynamic return ⇒ stack is NOT unwound (like routine)
- A resumption handler is a corrective action so a computation can continue.

```
_Event E {}; // uC++ exception label
void f() {
    _Resume E(); // raise
    // control returns here
}
void uMain::main() {
    try {
        f();
    } _CatchResume( E ) {
        cout << "handler 1" << endl;
    }
    try {
        f();
    } _CatchResume( E ) {
        cout << "handler 2" << endl;
    }
}
```

```
void f( void (*fixup)() ) {
    fixup()
    // control returns here
}
void fixup1() {
    cout << "handler 1" << endl;
}
void fixup2() {
    cout << "handler 2" << endl;
}
int main() {
    f( fixup1 );
    f( fixup2 );
}
```

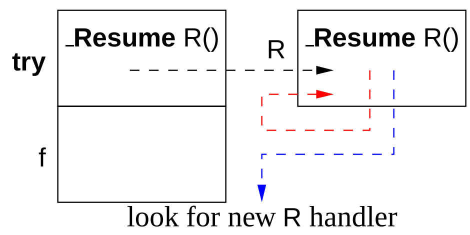
- Values at raise are modified via reference/pointer in caught exception:

```
_Event E {
  public:
    int &r;                 // reference to something
    E( int &r ) : r( r ) {}
};
void f() {
    int x;
    . . . _Resume E( x ); . . . // set exception reference to point to x
}
void g() {
    try {
        f();
    } _CatchResume( E &e ) {
        . . . e.r = 3; . . .      // change x at raise via reference r
    }
}
```

- No **break**, **continue**, **return** in **_CatchResume** handler (suppose to return).
- If a correction is impossible, the resumption handler should throw an exception not step into enclosing block to cause stack to unwind.
- May be recovery actions closer to raise point better able to handle problem.

- While a resumption handler remains on the stack, once it catches an exception, it is not *reused* until it has completed.
  - i.e., should the handler raise the same exception, it cannot catch it again because it has not fixed the previous problem.
  - hence, propagation ignores unfinished resumption handlers when looking for a handler
  - ⇒ no unbounded recursion:

```
struct R {};
void f() {
    try {
        _Resume R();
    } _CatchResume( R ) {
        . . . _Resume R(); . . .
    }
}
```



2.9 Implementation

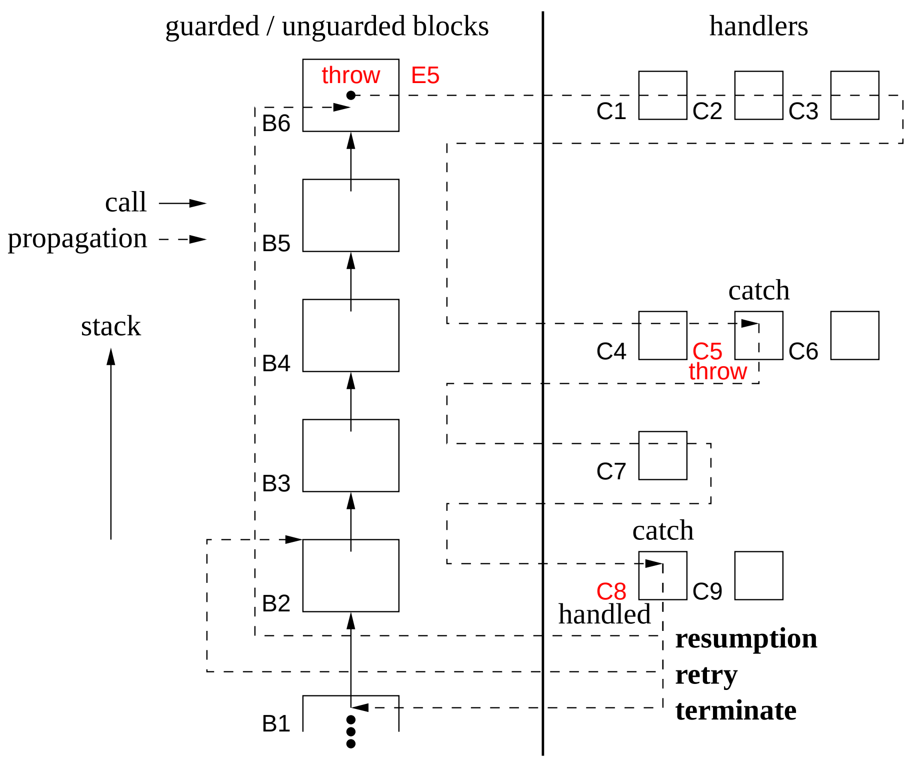
- To implement termination/resumption, the raise must know the last guarded block with a handler for the raised exception type.
- One approach is to:
  - associate a label variable with each exception type
  - set label variable on entry to each guarded block with handler for the type
  - reset label variable on exit to previous value, i.e., previous guarded block for that type
- For termination, a direct transfer is often impossible because
  - activations on the stack may contain objects with destructors or finalizers
  - hence, linear stack unwinding is necessary.
- For resumption, stack is not unwound, so direct transfer (call) to handler is possible.
- However, setting/resetting label variable on **try** block entry/exit is expensive:
  - rtn called million times but exception E never raised ⇒ million unnecessary operations.

```
void rtn( int i ) {
    try {                    // set label on entry
        . . .
    } catch( E ) { . . . }   // reset label on exit
}
```

  - Instead, **catch**/destructor data is stored once externally for each block and handler found by linear search during a stack walk (no direct transfer).
  - Advantage, millions of **try** entry/exit, but only tens of exceptions raised.
- Hence, both termination and resumption are often implemented using an expensive approach on raise and zero cost on guarded-block entry.

2.10 Exceptional Control-Flow

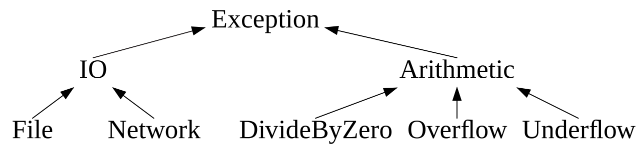
```
B1  {
B2        try {
B3              try {
B4                    try {
B5                          {
B6                                try {
                                        ... throw E5(); ...
C1                                } catch( E7 ) { ... }
C2                                  catch( E8 ) { ... }
C3                                  catch( E9 ) { ... }
                            }
C4                    } catch( E4 ) { ... }
C5                      catch( E5 ) { ... throw; ... }
C6                      catch( E6 ) { ... }
C7              } catch( E3 ) { ... }
C8        } catch( E5 ) { ... resume/retry/terminate }
C9          catch( E2 ) { ... }
    }
```



2.11 Additional features

2.11.1 Derived Exception-Type

- **derived exception-types** is a mechanism for inheritance of exception types, like inheritance of classes.
- Provides a kind of polymorphism among exception types:



- Provides ability to handle an exception at different degrees of specificity along the hierarchy.
- Possible to catch a more general exception-type in higher-level code where the implementation details are unknown.
- Higher-level code should catch general exception-types to reduce tight coupling to the specific implementation.
  - tight coupling may force unnecessary changes in the higher-level code when low-level code changes.
- Exception-type inheritance allows a handler to match multiple exceptions, e.g., a base handler can catch both base and derived exception-type.
- To handle this case, most propagation mechanisms perform a linear search of the handlers for a guarded block and select the first matching handler.

```
try { ...
} catch( Arithmetic ) { ...
} catch( Overflow ) { ... // never selected!!!
}
```

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

```
struct B {};
struct D : public B {};
try {
    throw D(); // _Throw in uC++
} catch( B e ) { // truncation
    // cannot down-cast
}
```

```
try {
    throw D(); // _Throw in uC++
} catch( B &e ) { // no truncation
    ... dynamic_cast<D>(e) ...
}
```

  - Otherwise, exception is truncated from its dynamic type to static type specified at the handler, and cannot be down-cast to the dynamic type.

2.11.2 Catch-Any

- **catch-any** is a mechanism to match any exception propagating through a guarded block.
- For termination, catch-any is used as a general cleanup when a non-specific exception occurs.
- For resumption, this capability allows a guarded block to gather or generate information about control flow.
- With exception-type inheritance, catch-any can be provided by the root exception-type, e.g., **catch**(Exception) in Java.

- Otherwise, special syntax is needed, e.g., **catch**(...) in C++.
- Java finalization:

```
try { ...
} catch( E ) { ... }
... // other catch clauses
} finally { ... } // always executed
```

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

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

2.11.3 Exception Parameters

- **exception parameters** allow passing information from the raise to a handler.
- Inform a handler about details of the exception, and to modify the raise site to fix an exceptional situation.
- Different EHMs provide different ways to pass parameters.
- In C++/Java, parameters are defined inside the exception:

```
struct E {
    int i;
    E( int i ) : i(i) {}
};
void f(...) { ... throw E( 3 ); ... } // argument
int main() {
    try {
        f(...);
    } catch( E p ) { // parameter
        // use p.i
    }
}
```

2.11.4 Exception List

- Missing exception handler for arithmetic overflow in control software caused Ariane 5 rocket to self-destruct ($370 million loss).
- **exception list** is part of a routine's prototype specifying which exception types may propagate from the routine to its caller.

```
int g() throw(E) { ... throw E(); }
```

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

- 2 kinds of checking:
  - checked/unchecked exception-type (Java, inheritance based, static check)
  - checked/unchecked routines (C++, exception-list based, dynamic check)
    (deprecated C++11, replaced with **noexcept**)
- While checked exception-types are useful for software engineering, reuse is precluded.
- E.g., consider the simplified C++ template routine sort:

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

  using the operator routine < in its definition.

- Impossible to know all exception types that propagated from routine < for every type.
- Since only a fixed set of exception types can appear in sort's exception list, some sortable types are precluded.
- Exception lists can preclude reuse for arguments of routine pointers (functional style) and/or polymorphic methods/routines (OO style):

```
// throw NO exceptions
void f( void (*p)() throw() ) {
    p();
}
void g() throw(E) { throw E(); }
void h() {
    try { . . . f( g ); . . .
    } catch( E ) {}
}
```

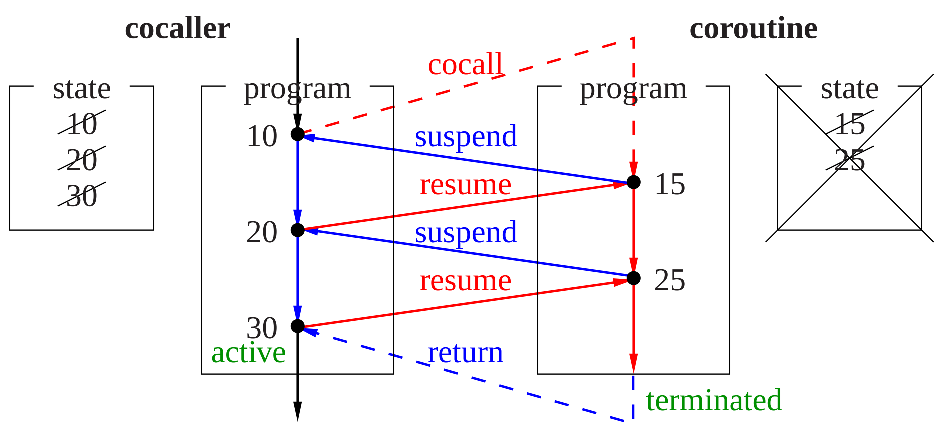
```
struct B { // throw NO exceptions
    virtual void g() throw() {}
    void f() { g(); }
};
struct D : public B {
    void g() throw(E) { throw E(); }
    void h() {
        try { . . . f(); . . .
        } catch( E ) {}
    }
};
```

- Left example, routine h has an appropriate **try** block and passes the version of g to f that raises exception-type E.
- However, checked exception-types preclude this case because the signature of argument g is less restrictive than parameter p of f.
- Right example, member routine D::h calls B::f, which calls D::g that raises exception-type E.
- However, checked exception types preclude this case because the signature of D::g is less restrictive than B::g.
- Finally, determining an exception list for a routine can become impossible for concurrent exceptions because they can propagate at any time.

3 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.

3.1 Semi-Coroutine

3.1.1 Fibonacci Sequence

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

- 3 states, producing the sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

3.1.1.1 Direct

- Compute and print Fibonacci numbers.

```
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):

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

- Examine different solutions.

3.1.1.2 Routine

```
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

3.1.1.3 Class

```
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
}
```

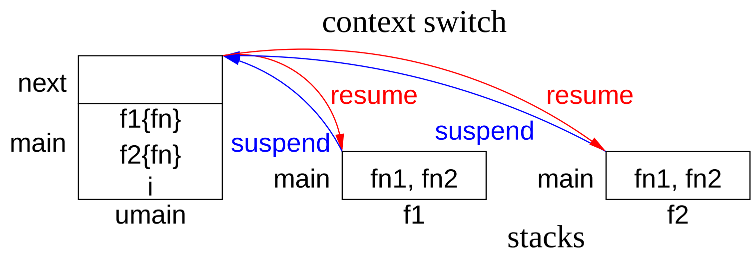
- unencapsulated program global variables become encapsulated object global variables
- multiple fibonacci generators (objects) can run at a time
- execution state must still be explicitly retained

3.1.1.4 Coroutine

```
#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;
    }
    uRetCode = 3;        // optional, return code
}
```

- **no explicit execution state!** (see direct solution)
- distinguished member main (coroutine main) can be suspended and resumed
- no parameters or return value (supplied by **public** members and communication variables).
- main can be called (even recursively), but normally a **private**/**protected** member. Why?

- first resume starts main on new stack (cocall); subsequent resumes reactivate last suspend.
- suspend reactivates last resume
- object becomes a coroutine on first resume; coroutine becomes an object when main ends
- both statements cause a **context switch** between coroutine stacks



- routine frame at the top of the stack *knows* where to activate execution
- suspend/resume are **protected** members to prevent external calls. Why?
- Coroutine main does not have to return before a coroutine object is deleted.
- When deleted, a coroutine's stack is always unwound and any destructors executed. Why?
- uMain is the initial coroutine started by μC++
- argc, argv, and uRetCode are implicitly defined in uMain::main
- compile with u++ command

3.1.2 Format Output

Unstructured input:

```
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Structured output:

```
abcd  efgh  ijkl  mnop  qrst
uvwx  yzab  cdef  ghij  klmn
opqr  stuv  wxyz
```

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

3.1.2.1 Direct

- Read characters and print formatted output.

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

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

3.1.2.2 Routine

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

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

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

3.1.2.3 Class

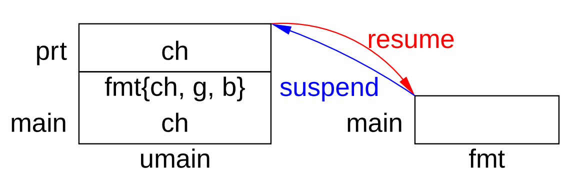
```
class Format {
    int g, b;                         // global class variables
  public:
    Format() : g( 0 ), b( 0 ) {}
    ~Format() { if ( g != 0 || b != 0 ) cout << endl; }
    void prt( char ch ) {
      if ( ch == '\n' ) return;       // ignore newline
        cout << ch;                   // print character
        b += 1;
        if ( b == 4 ) {               // block of 4 chars
            cout << "  ";             // block separator
            b = 0;
            g += 1;
        }
        if ( g == 5 ) {               // group of 5 blocks
            cout << endl;             // group separator
            g = 0;
        }
    }
};
int main() {
    Format fmt;
    char ch;
    cin >> noskipws;                  // turn off white space skipping
    for ( ;; ) {                      // for as many characters
        cin >> ch;                    // read one character
      if ( cin.fail() ) break;        // eof ?
        fmt.prt( ch );
    }
}
```

- Solves encapsulation and multiple instances issues, but explicitly managing execution state.

3.1.2.4 Coroutine

```
_Coroutine Format {
    char ch;                        // used for communication
    int g, b;                       // global because used in destructor
    void main() {
        for ( ;; ) {                            // for as many characters
            for ( g = 0; g < 5; g += 1 ) {  // groups of 5 blocks
                for ( b = 0; b < 4; b += 1 ) {  // blocks of 4 characters
                    for ( ;; ) {            // for newline characters
                        suspend();
                      if ( ch != '\n' ) break; // ignore newline
                    }
                    cout << ch;             // print character
                }
                cout << "  ";               // print block separator
            }
            cout << endl;                   // print group separator
        }
    }
  public:
    Format() { resume(); }              // start coroutine
    ~Format() { if ( g != 0 || b != 0 ) cout << endl; }
    void prt( char ch ) { Format::ch = ch; resume(); }
};
void uMain::main() {
    Format fmt;
    char ch;
    cin >> noskipws;                    // turn off white space skipping
    for ( ;; ) {
        cin >> ch;                      // read one character
      if ( cin.fail() ) break;          // eof ?
        fmt.prt( ch );
    }
}
```

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



3.1.3 Correct Coroutine Usage

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

BAD: Explicit Execution State	GOOD: Implicit Execution State

``for ( int i = 0; i < 10; i += 1 ) {``
``    if ( i % 2 == 0 ) // even ?``
``        even += digit;``
``    else``
``        odd += digit;``
``    suspend();``
``}``
	``for ( int i = 0; i < 5; i += 1 ) {``
	
	``    even += digit;``
	``    suspend();``
	``    odd += digit;``
	``    suspend();``
	``}``

- Right example illustrates coroutine "Zen"; let it do the work.
- E.g., a BAD solution for the previous Fibonacci generator is:

```
void main() {
    int fn1, fn2, state = 1;
    for ( ;; ) {
        switch (state) {            // no Zen
          case 1:
            fn = 0;  fn1 = fn;
            state = 2;
            break;
          case 2:
            fn = 1;  fn2 = fn1;  fn1 = fn;
            state = 3;
            break;
          case 3:
            fn = fn1 + fn2;  fn2 = fn1;  fn1 = fn;
            break;
        }
        suspend();                  // no Zen
    }
}
```

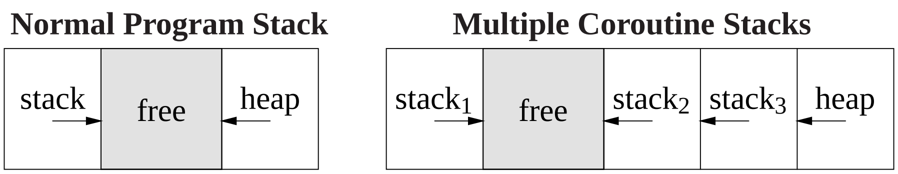
- Coroutine's capabilities not used:
  - explicit flag variable controls execution state
  - original program structure lost in **switch** statement
- Must do more than just *activate* coroutine main to demonstrate understanding of retaining data and execution state within a coroutine.

3.1.4 Coroutine Construction

- Fibonacci and formatter coroutines express original algorithm structure (no restructuring).
- When possible, simplest coroutine construction is to write a direct (stand-alone) program.
- Convert to coroutine by:
  - putting processing code into coroutine main,
  - converting reads if program is consuming or writes if program is producing to suspend,

    * Fibonacci consumes nothing and produces (generates) Fibonacci numbers ⇒ convert writes (cout) to suspends.
    * Formatter consumes characters and only indirectly produces output (as side-effect) ⇒ convert reads (cin) to suspends.
  - use interface members and communication variables to transfer data in/out of coroutine.

- Memory management



  - Normally program stack expands to heap; but coroutine stacks expand to next stack.
  - In fact, coroutine stacks are normally allocated in the heap.
  - Default μC++ coroutine stack size is 64K **and it does not grow**.
  - Adjust coroutine stack-size through coroutine constructor:

```
_Coroutine C {
  public:
    C() : uBaseCoroutine( 8192 ) {};      // default 8K stack
    C( int size ) : uBaseCoroutine( size ) {};  // user specified stack size
    . . .
};
C x, y( 16384 );      // x has an 8K stack, y has a 16K stack
```

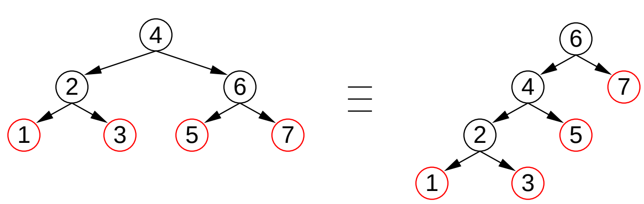
  - Check for stack overflow using coroutine member verify:

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

  - Be careful allocating arrays in the coroutine main; sometimes necessary to allocate large arrays in heap.

3.1.5 Same Fringe

- Two binary trees have same fringe if all leafs are equals from left to right.



- Requires iterator to traverse a tree, return the value of each leaf, and continue the traversal.

- No direct solution without using additional data-structure (e.g., stack) to manage the tree traversal.
- Coroutine uses recursive tree-traversal but suspends traversal to return value.

```
template< typename T > class Btree {
    struct Node { . . . };
    . . .
  public:
    _Coroutine Iterator {
        Node *cursor;
        void walk( Node *node ) {   // walk tree
          if ( node == NULL ) return;
            if ( node->left() == NULL && node->right() == NULL ) { // leaf?
                cursor = node;
                suspend();          // multiple stack frames
            } else {
                walk( node->left );   // recursion
                walk( node->right);   // recursion
            }
        }
        void main() { walk( cursor ); }
      public:
        Iterator( Btree<T> &btree ) : cursor( &btree.root ) {}
        T *next() {
            if ( cursor != NULL ) resume();
            return cursor;
        }
    };
    . . .
};
template<class T> bool sameFringe( BTree<T> &tree1, BTree<T> &tree2 ) {
    Btree<T>::Iterator iter1( btree1 ), iter2( btree2 );
    T *t1, *t2;
    for ( ;; ) {
        t1 = iter1.next();  t2 = iter2.next();
        if ( t1 == NULL || t2 == NULL ) break; // one traversal complete ?
        if ( *t1 != *t2 ) return false; // elements equal ?
    }
    return t1 == NULL && t2Ptr == NULL; // and both traversals completed
}
```

3.1.6 Device Driver

- Called by interrupt handler for each byte arriving at hardware serial port.
- Parse transmission protocol and return message text, e.g.:

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

3.1.6.1 Direct

```
void uMain::main() {
    enum { STX = '\002', ESC = '\033', ETX = '\003' };
    enum { MaxMsgLnth = 64 };
    unsigned char msg[MaxMsgLnth];
    ...
    try {
      msg: for ( ;; ) {                         // parse messages
            int lnth = 0, checkval;
            do {
                byte = input( infile );          // read bytes, throw Eof on eof
            } while ( byte != STX );             // message start ?
          eom: for ( ;; ) {                      // scan message data
                byte = input( infile );



                switch ( byte ) {
                  case STX:
                    ...                          // protocol error
                    continue msg;                // uC++ labelled continue
                  case ETX:                      // end of message
                    break eom;                   // uC++ labelled break
                  case ESC:                      // escape next byte
                    byte = input( infile );
                    break;
                } // switch



                if ( lnth >= 64 ) {              // buffer full ?
                    ...                          // length error
                    continue msg;                // uC++ labelled continue
                } // if
                msg[lnth] = byte;                // store message
                lnth += 1;
            } // for
            byte = input( infile );              // gather check value
            checkval = byte;
            byte = input( infile );
            checkval = (checkval << 8) | byte;
            if ( ! crc( msg, lnth, checkval ) ) ... // CRC error
        } // for
    } catch( Eof ) {}
    ...
} // uMain
```

3.1.6.2 Coroutine

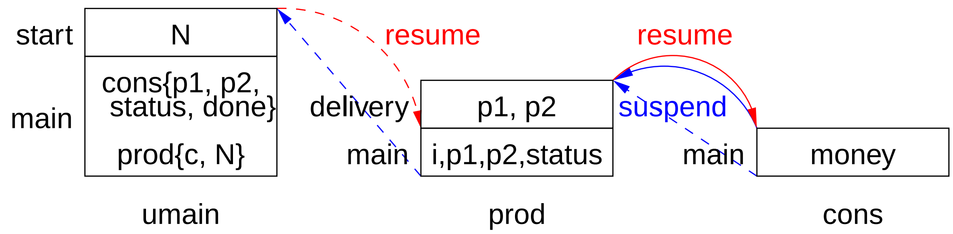
```
_Coroutine DeviceDriver {
    enum { STX = '\002', ESC = '\033', ETX = '\003' };
    unsigned char byte;
    unsigned char *msg;
  public:
    DeviceDriver( unsigned char *msg ) : msg( msg ) { resume(); }
    void next( unsigned char b ) { // called by interrupt handler
        byte = b;
        resume();
    }
  private:
    void main() {
      msg: for ( ;; ) {                       // parse messages
            int lnth = 0, checkval;
            do {
                suspend();
            } while ( byte != STX );          // message start ?
          eom: for ( ;; ) {                   // scan message data
                suspend();

                switch ( byte ) {
                  case STX:
                    ...                       // protocol error
                    continue msg;             // uC++ labelled continue
                  case ETX:                   // end of message
                    break eom;                // uC++ labelled break
                  case ESC:                   // escape next byte
                    suspend();                 // get escaped character
                    break;
                } // switch

                if ( lnth >= 64 ) {           // buffer full ?
                    ...                       // length error
                    continue msg;             // uC++ labelled continue
                } // if
                msg[lnth] = byte;             // store message
                lnth += 1;
            } // for

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

3.1.7 Producer-Consumer

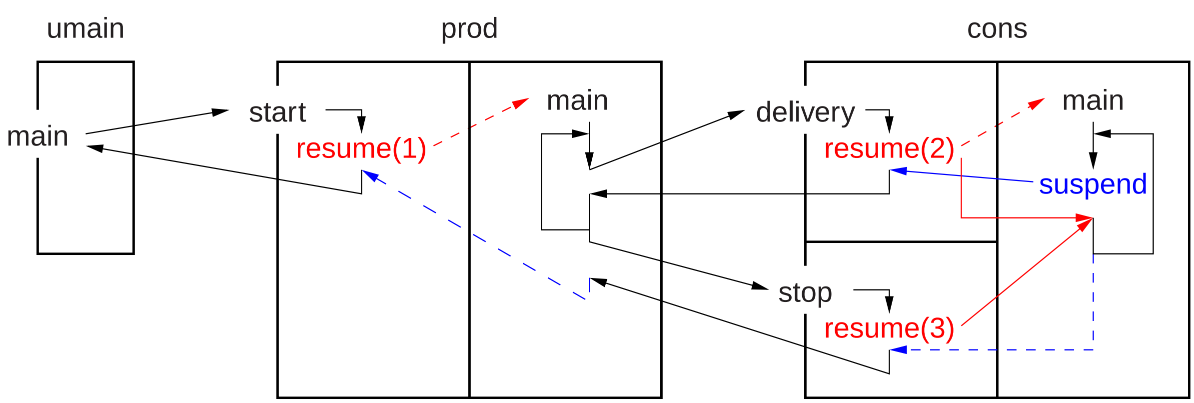
```
_Coroutine Cons {
    int p1, p2, status;  bool done;
    void main() { // starter prod
        // 1st resume starts here
        int money = 1;
        for ( ;; ) {
          if ( done ) break;
            cout << "receives:" << p1 << ", " << p2;
            cout << " and pays $" << money << endl;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

```
_Coroutine Prod {
    Cons &c;
    int N;
    void main() { // starter umain
        // 1st resume starts here
        int i, p1, p2, status;
        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)
```

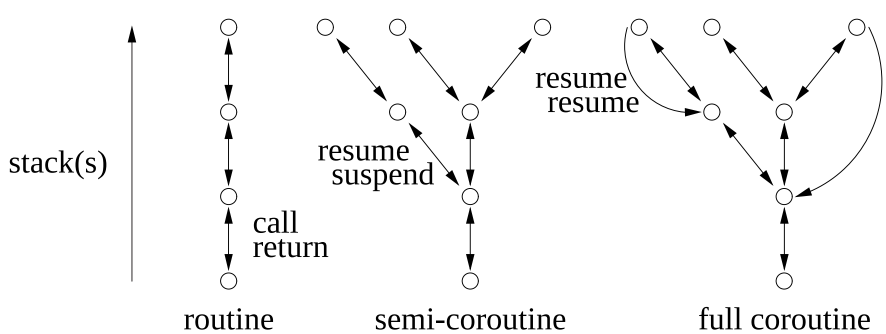


- Do both Prod and Cons need to be coroutines?
- When coroutine main returns, it activates the coroutine that *started* main.
- prod started cons.main, so control goes to prod suspended in stop.
- uMain started prod.main, so control goes back to uMain suspended in start.



3.2 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.

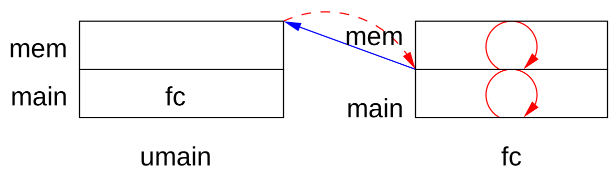


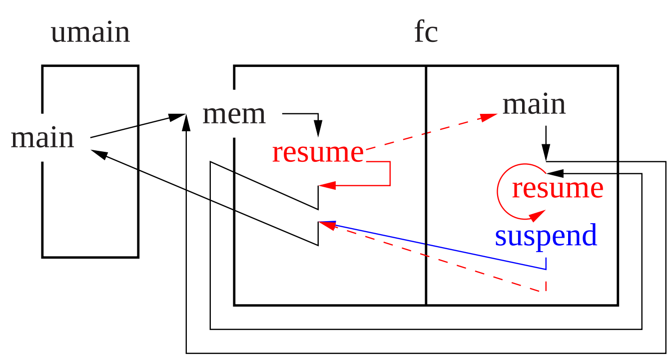
- A full coroutine is allowed to perform semi-coroutine operations because it subsumes the notion of semi-routine.

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

control flow semantics

	inactive	active
uThisCoroutine()	resume	this
	suspend	last resumer
	context switch	





- Suspend inactivates the current active coroutine (uThisCoroutine), and activates last resumer.
- Resume inactivates the current active coroutine (uThisCoroutine), and activates the current object (**this**).
- Hence, the current object *must* be a non-terminated coroutine.
- Note, **this** and 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:

```
Fc x(y), y(x);
```

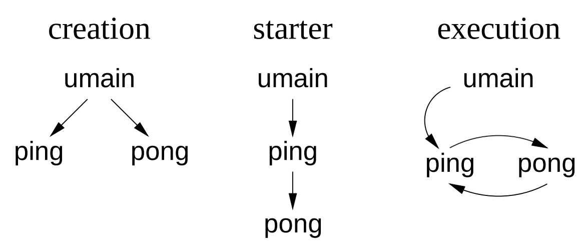
- One solution is to make closing the cycle a special case:

```
Fc x, y(x);
x.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***.
- A **starter** coroutine is the coroutine that does the first resume (cocall).
- For semi-coroutines, the starter is often the last (only) resumer, so it seems coroutine main implicitly suspends on termination.
- For full-coroutines, the starter is often *not* the last resumer, so it does *not* seem coroutine main implicitly suspends on termination.
- In many cases, it is unnecessary to terminate all coroutines, just delete them.
- But it is necessary to activate uMain for the program to finish (unless exit is used).

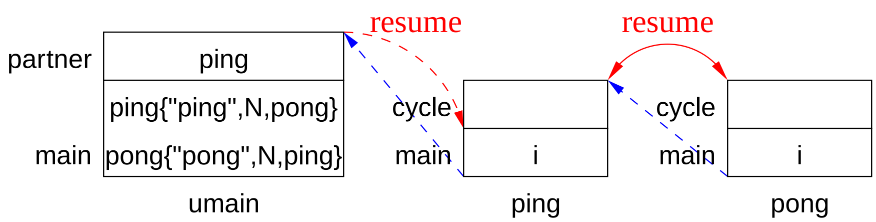
3.2.1 Ping/Pong

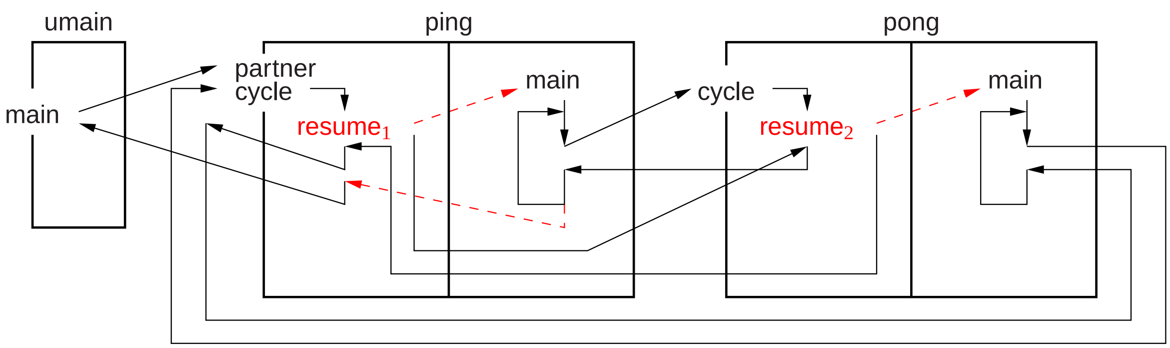
- Full-coroutine control-flow: 2 identical coroutines:



```
_Coroutine PingPong {
    const char *name;
    const unsigned int N;
    PingPong *part;
    void main() { // ping's starter umain, pong's starter ping
        for ( unsigned int i = 0; i < N; i += 1 ) {
            cout << name << endl;
            part->cycle();
        }
    }
  public:
    PingPong( const char *name, unsigned int N, PingPong &part )
        : name( name ), N( N ), part( &part ) {}
    PingPong( const char *name, unsigned int N ) : name( name ), N( N ) {}
    void cycle() { resume(); }
    void partner( PingPong &part ) { PingPong::part = &part; resume(); }
};
void uMain::main() {
    enum { N = 20 };
    PingPong ping( "ping", N ), pong( "pong", N, ping );
    ping.partner( pong );
}
```

- ping created without partner; pong created with partner.
- ping makes pong partner, closing cycle.
- Why is PingPong::part a pointer rather than reference?
- partner resumes ping ⇒ umain is ping's starter
- ping calls pong's cycle member, which resumes pong so ping is pong's starter.
- pong calls ping's cycle member, **which resumes ping in pong's cycle member**.
- Each coroutine cycles N times, **becoming inactive in the other's cycle member**.
- ping ends first, because it started first, implicitly resuming its starter umain in ping's partner member.
- umain terminates with terminated coroutine ping and unterminated coroutine pong.
- Assume ping's declaration is changed to ping("ping", N + 1).
- pong ends first, implicitly resuming its starter ping in pong's cycle member.
- ping ends second, implicitly resuming its starter umain in ping's partner member.
- umain terminates with terminated coroutines ping and pong.





3.2.2 Producer-Consumer

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

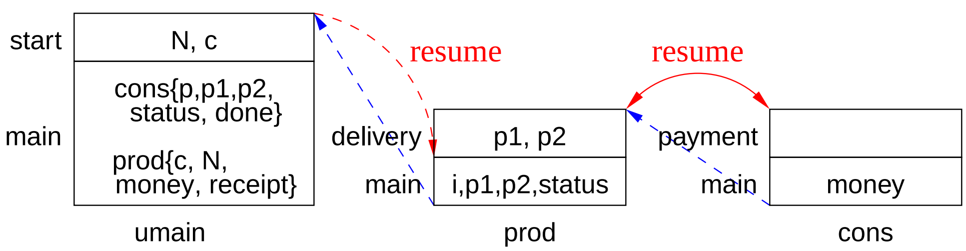
```
_Coroutine Prod {
    Cons *c;
    int N, money, receipt;
    void main() { // starter umain
        // 1st resume starts here
        for ( int i = 0; i < N; i += 1 ) {
            int p1 = rand() % 100;
            int p2 = rand() % 100;
            cout << p1 << " " << p2 << endl;
           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

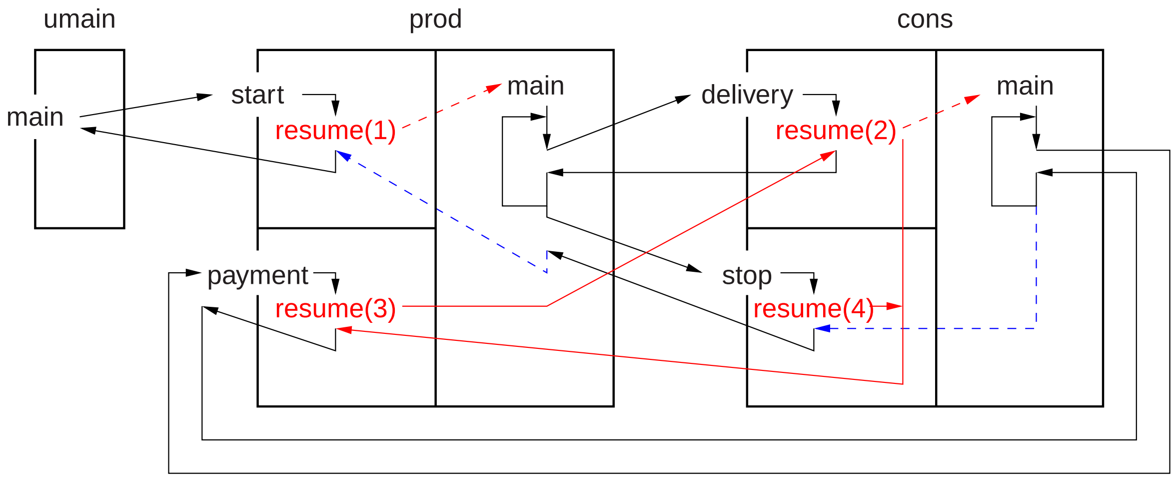
```
  public:
    int payment( int money ) {
        Prod::money = money;
        resume(); // Prod::main 1st time, then
        return receipt; // prod in Cons::delivery
    }
    void start( int N, Cons &c ) {
        Prod::N = N; Prod::c = &c;
        receipt = 0;
        resume(); // activate main
    }
};
```

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

```
  public:
    Cons( Prod &p ) : p(p), done(false) {}
    int delivery( int p1, int p2 ) {
        Cons::p1 = p1; Cons::p2 = p2;
        resume(); // Cons::main 1st time, then
        return status; // cons in Prod::payment
    }
    void stop() {
        done = true;
        resume(); // activate Prod::payment
    }
};
void uMain::main() {
    Prod prod;
    Cons cons( prod );
    prod.start( 5, cons );
}
```

- Cheat using forward reference for Prod at p.payment. Fix by?





3.3 Coroutine Languages

- Coroutine implementations have two forms:

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

- Stackless coroutines cannot call other routines and then suspend, i.e., only suspend in the coroutine main.
- Generators/iterators are often simple enough to be stackless using yield.
- Simula, CLU, C#, Ruby, Python, JavaScript, Lua, F#, Boost all support yield constructs.

3.3.1 Python 3.4.1

- Stackless, semi coroutines, routine versus class, no calls, single interface
- Fibonacci (see Section 3.1.1.4, p. 26)

```
def Fibonacci( n ):                 # coroutine main
    fn = 0;  fn1 = fn
    yield fn                        # suspend
    fn = 1; fn2 = fn1; fn1 = fn
    yield fn                        # suspend
    # while True:                   # for infinite generator
    for i in range( n - 2 ):
        fn = fn1 + fn2; fn2 = fn1; fn1 = fn
        yield fn                    # suspend

f1 = Fibonacci( 10 )                # objects
f2 = Fibonacci( 10 )
for i in range( 10 ):
    print next( f1 ), next( f2 )    # resume
for fib in Fibonacci( 15 ):         # use generator as iterator
    print fib
```

- Format (see Section 3.1.2.4, p. 30)

```
def Format():
    try:
        while True:
            for g in range( 5 ):        # groups of 5 blocks
                for b in range( 4 ):    # blocks of 4 characters
                    print( (yield), end='' ) # receive from send
                print( '  ', end='' )   # block separator
            print()                     # group separator
    except GeneratorExit:               # destructor
        if g != 0 | b != 0:             # special case
            print()

fmt = Format()
next( fmt )                             # prime generator
for i in range( 41 ):
    fmt.send( 'a' )                     # send to yield
```

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

3.3.2 C++ Boost Library

- stackfull, semi/full coroutines, routine versus class, no recursion, single interface
- **return** activates creator or raise an exception via exit
- full coroutines use yield_to(partner, ...), specifying target coroutine to be activated
- coroutine state passed as explicit parameter
- Fibonacci (semi-coroutine)

```
#include <boost/coroutine/coroutine.hpp>
using namespace boost::coroutines;
int fibonacci( coroutine<int()>::self &self ) {
    int fn, fn1, fn2;
    fn = 0; fn1 = fn;
    self.yield( fn );
    fn = 1; fn2 = fn1; fn1 = fn;
    self.yield( fn );
    for ( ;; ) {
        fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
        self.yield( fn );
    }
}
int main() {
    coroutine<int()> fib( fibonacci );
    for ( int i = 0; i < 30; i += 1 ) {
        cout << fib() << endl;
    }
}
```

- Producer/Consumer (full-coroutine)

```
#include <boost/bind.hpp>
#include <boost/coroutine/coroutine.hpp>
using namespace boost::coroutines;
typedef coroutine< int ( int, int ) > producer_type;
typedef coroutine< int ( int, int, int ) > consumer_type;
int Producer( producer_type::self &self,
    int money, int status, consumer_type &consumer ) {
    int receipt = 0;
    for ( int i = 0; i < 5; i += 1 ) {
        int p1 = rand() % 100, p2 = rand() % 100;
        cout << "delivers:" << p1 << ", " << p2 << endl;
        boost::tie( money, status ) = // return 2 values
            self.yield_to( consumer, p1, p2, receipt );
        cout << " gets payment $" << money << endl;
        cout << " gets status:" << status << endl;
        receipt += 1;
    }
    cout << "Prod stops" << endl;
    self.yield_to( consumer, -1, -1, receipt );
}
```

```
int Consumer( consumer_type::self &self, int p1, int p2,
        int receipt, producer_type &producer ) {
    int money = 1, status = 0;
    for ( int i = 0; i < 5; i += 1 ) {
        cout << "receives:" << p1 << ", " << p2 <<
            " and pays $" << money << endl;
        status += 1;
        boost::tie( p1, p2, receipt ) = // return 2 values
            self.yield_to( producer, money, status );
        cout << "gets receipt #" << receipt << endl;
        money += 1;
    }
    cout << "Cons stops" << endl;
    self.yield_to( producer, -1, -1 );
}

int main() {
    producer_type producer;
    consumer_type consumer;
    producer = producer_type( boost::bind( Producer,
                    _1, _2, _3, boost::ref( consumer ) ) );
    consumer = consumer_type( boost::bind( Consumer,
                    _1, _2, _3, _4, boost::ref( producer ) ) );
    producer( 0, 0 );
}
```

4 μC++ EHM

The following features characterize the μC++ EHM:

- μC++ exceptions are generated from a specific kind of type, which can be thrown and/or resumed.
  - All exception types are grouped into a hierarchy.
  - μC++ provides a set of predefined exception-types covering exceptional runtime and I/O events.
- μC++ restricts raising to a specific exception type.
- μC++ supports two forms of raising, throwing and resuming.
- μC++ supports two kinds of handlers, termination and resumption, which match with the kind of raise.
- μC++ supports propagation of nonlocal and concurrent exceptions.

4.1 Exception Type

- C++ allows any type to be used as an exception type. μC++ restricts exception types to those types defined by **_Event**.

```
_Event exception-type-name {
    . . .
};
```

- An exception type has all the properties of a **class**.
- As well, every exception type must have a public default and copy constructor.
- An exception is the same as a class-object with respect to creation and destruction:

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

4.2 Inherited Members

- Each exception type inherits the following members from uBaseEvent:

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

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

  message is copied so it is safe to use within an exception even if the context of the raise is deleted.

- message returns the string message associated with an exception.
- source returns the coroutine/task that raised the exception.

  coroutine/task may be deleted when the exception is caught so this reference may be undefined.

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

  name is copied from the raising coroutine/task when exception is created.

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

  default action is to terminate the program with the supplied message.

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

  default action is to throw the exception.

4.3 Raising

- There are two raising mechanisms, throwing and resuming.

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

- If **_Throw** has no *exception-type*, it is a rethrow.
- If **_Resume** has no *exception-type*, it is a reresume.
- The optional **_At** clause allows the specified exception or the currently propagating exception to be raised at another coroutine or task.
- Nonlocal/concurrent raise restricted to resumption as raising execution-state is often unaware of the handling execution-state.
- Resumption allows faulting execution greatest flexibility: it can process the exception as a resumption or rethrow the exception for termination.

- Exceptions in μC++ are propagated differently from C++.

C++	μC++

<pre>**class** B {};
class D : **public** B {};
void f(B &t) {
    throw t;
}
D m;
f(m);</pre>
	<pre>**_Event** B {};
	_Event D : **public** B {};
	void f(B &t) {
	    _Throw t;
	}
	D m;
	f(m);</pre>

  - In C++, routine f is passed an object of derived type D but throws an object of base type B.
  - In μC++, routine f is passed an object of derived type D and throws the original object of type D.
  - This change allows handlers to catch the specific (derived) rather than the general (base) exception-type.

- Notice, catching truncation (see page 20) is different from raising truncation, which does not occur in μC++.

4.4 Handler

- μC++ has two kinds of handlers, termination and resumption, which match with the kind of raise.

4.4.1 Termination

- The μC++ termination handler is the **catch** clause of a **try** block, i.e., same as in C++.

4.4.2 Resumption

- A resumption handler is often a corrective action for a failing operation.
- Unlike normal routine calls, the call to a resumption handler is dynamically bound rather than statically bound.
- μC++ extends the **try** block to include resumption handlers.
- Resumption handler is denoted by a **_CatchResume** clause at the end of a **try** block:

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

- Any number of resumption handlers can be associated with a **try** block.
- All **_CatchResume** handlers must precede any **catch** handlers.
- Like **catch**(...) (catch-any), **_CatchResume**(...) must appear at the end of the list of the resumption handlers.
- Resumption handler can access types and variables visible in its local scope.
- ***It cannot perform a* break*,* continue *or* return *from within the handler.***
  - A resumption handler is a corrective action so a computation can continue.
  - If correction impossible, handler should throw an exception not step into an enclosing block to cause the stack to unwind.

```
B: {
    try {
        f(); // recursive calls and _Resume E()
    } _CatchResume( E ) { // handler H
        . . . goto L;  // force static return
    }
  L: . .
}
```

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

- Values at the raise site can be modified indirectly through reference or pointer variables in the caught exception:

```
_Event E {
  public:
    int &r;                    // reference to something
    E( int &r ) : r( r ) {}
};
void f() {
    int x;
    . . . _Resume E( x ); . . . // set exception reference to point to x
}
void g() {
    try {
        f();
    } _CatchResume( E &e ) {
        . . . e.r = 3; . . .        // change x at raise via reference r
    }
}
```

- μC++ uses marking to prevent recursive resuming.

4.4.3 Termination/Resumption

- The form of the raise dictates the set of handlers examined during propagation:
  - terminating propagation (**_Throw**) only examines termination handlers (**catch**),
  - resuming propagation (**_Resume**) only examines resumption handlers (**_CatchResume**).
- The set of exception types in each set can overlap:

```
_Event E {};
void rtn() {
    try {
        _Resume E();
    }   _CatchResume( E & ) { _Throw E(); }  // H1
        catch( E & ) { ... }                 // H2
}
```

- Resumption handler H1 is invoked by the resume in the **try** block generating call stack:

  rtn → **try**{}**_CatchResume(E)**, **catch**(E)→ H1

- Handler H1 throws E and the stack is unwound until the exception is caught by termination-handler **catch**(E) and handler H2 is invoked.

  rtn → H2

- The termination handler is available because resuming did not unwind the stack.
- Note the interaction between resuming, defaultResume, and throwing:

```
_Event R {};
void rtn() {
    try {
        _Resume R();        // resume not throw
    } catch( R & ) { ... }  // H1, no _CatchResume!!!
}
```

- This generates the following call stack as there is no eligible resumption handler (or there is a handler but marked ineligible):

  rtn → **try**{}**catch(R)** → defaultResume

- ***When* defaultResume *is called, the default action throws* R** (see Section 4.2, p. 47).

  rtn → H1

- Terminating propagation unwinds the stack until there is a match with the **catch** clause in the **try** block.

4.5 Nonlocal Exceptions

- 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 {};
    _Coroutine C {
        void main() { _Throw E(); }
      public:
        void mem() { resume(); }
    };
    void uMain::main() {
        C c;
        try { c.mem();
        } _CatchResume( 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.*

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

- Nonlocal exceptions are possible because each coroutine has its own stack.

- **Nonlocal delivery is initially disabled for a coroutine**, so handlers can be set up before any exception can be delivered (also see Section 5.14, p. 68).

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

```
_Event E {};
_Coroutine C {
    void main() {
        try {
            _Enable {   // allow nonlocal exceptions
                ... suspend(); ...
            }           // disable all nonlocal exceptions
        } catch( E ) { ... }
    }
  public:
    C() { resume(); }   // prime loop
    void mem() { resume(); }
};
void uMain::main() {
    C c;
    _Resume E() _At c; // exception pending
    c.mem();            // trigger exception
}
```

- 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.

- **For nonlocal resumption, _Resume is a *proxy* for actual raise in the faulting coroutine.**

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

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

- *μ*C++ allows dynamic enabling and disabling of individual exception types versus all exception types.

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

- Specifying no exceptions is shorthand for specifying 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.

5 Concurrency

- A **thread** is an independent sequential execution path through a program. Each thread is scheduled for execution separately and independently from other threads.
- A **process** is a program component (like a routine) that has its own thread and has the same state information as a coroutine.
- A **task** is similar to a process except that it is reduced along some particular dimension (like the difference between a boat and a ship, one is physically smaller than the other). It is often the case that a process has its own memory, while tasks share a common memory. A task is sometimes called a light-weight process (LWP).
- **Parallel execution** is when 2 or more operations occur simultaneously, which can only occur when multiple processors (CPUs) are present.
- **Concurrent execution** is any situation in which execution of multiple threads *appears* to be performed in parallel. It is the threads of control associated with processes and tasks that results in concurrent execution.

5.1 Why Write Concurrent Programs

- Dividing a problem into multiple executing threads is an important programming technique just like dividing a problem into multiple routines.
- Expressing a problem with multiple executing threads may be the natural (best) way of describing it.
- Multiple executing threads can enhance execution-time efficiency by taking advantage of inherent concurrency in an algorithm and any parallelism available in the computer system.

5.2 Why Concurrency is Difficult

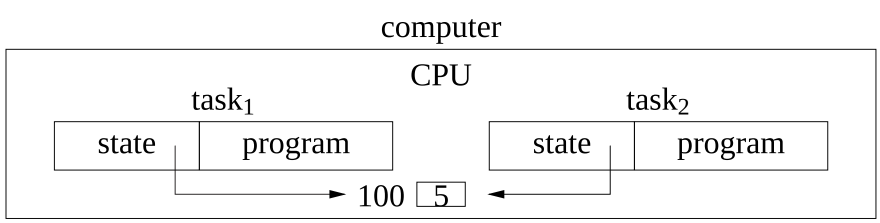
- to understand:
  - While people can do several things concurrently, the number is small because of the difficulty in managing and coordinating them.
  - Especially when the things interact with one another.
- to specify:
  - How can/should a problem be broken up so that parts of it can be solved at 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

5.3 Concurrent Hardware

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

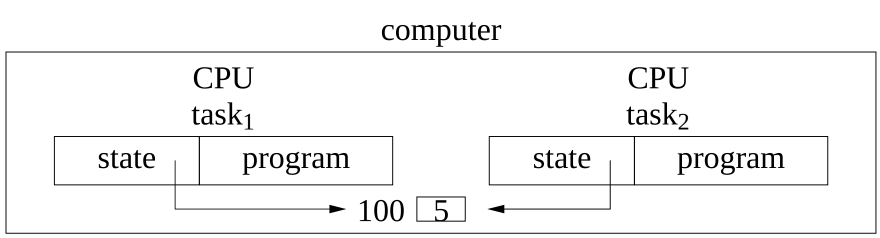


    - Parallelism is simulated by context switching the threads on the CPU.
    - **Unlike coroutines, task switching may occur at non-deterministic program locations, i.e., between any two *machine* instructions.**
    - Introduces all the difficulties in concurrent programs.
        - programs must be written to work regardless of non-deterministic ordering of program execution.
    - Switching happens *explicitly* but conditionally when calling routines.
        - routine may or may not context switch depending on hidden (internal) state (cannot predict)

- ◦ Switching can happen *implicitly* because of an external **interrupt** independent of program execution.
  - ∗ e.g., I/O or timer interrupt;
  - ∗ timer interrupts divide execution (between instructions) into discrete time-slices occurring at non-deterministic time intervals
  - ∗ ⇒ task execution is not continuous
- ◦ If interrupts affect **scheduling** (execution order), it is called **preemptive**, otherwise the scheduling is **non-preemptive**.
- ◦ Programmer cannot predict execution order, unlike coroutines.
- ◦ Granularity of context-switch is instruction level for preemptive (harder to reason) and routine level for non-preemptive.
- ◦ Pointers among tasks work because memory is shared.
- ◦ Most of the issues in concurrency can be illustrated without parallelism.

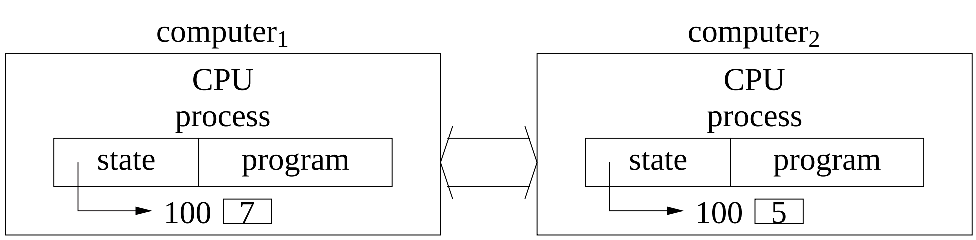
- In fact, every computer has multiple CPUs: main CPU(s), bus CPU, graphics CPU, disk CPU, network CPU, etc.

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



- Pointers among tasks work because memory is shared.

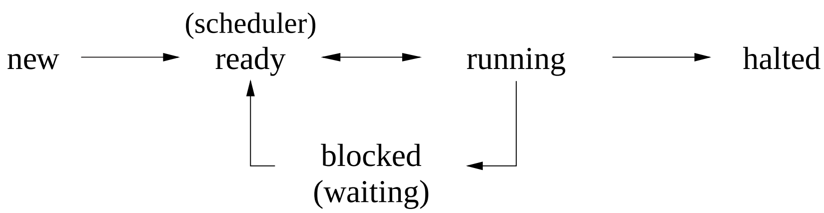
- Concurrent/parallel execution of threads is possible with single/multiple CPUs on different computers with *separate memories* (**distributed system**):



- Pointers among tasks do NOT work because memory is not shared.

5.4 Execution States

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



- **state transitions** are initiated in response to events (interrupts):
  - timer alarm (running → ready)
  - completion of I/O operation (blocked → ready)
  - exceeding some limit (CPU time, etc.) (running → halted)
  - error (running → halted)
- non-deterministic "ready ↔ running" transition ⇒ basic operations unsafe:

```
int i = 0;
task0         task1
i += 1        i += 1
```

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

```
ld   r1,i      // load into register 1 the value of i
add  r1,#1     // add 1 to register 1
st   r1,i      // store register 1 into i
```

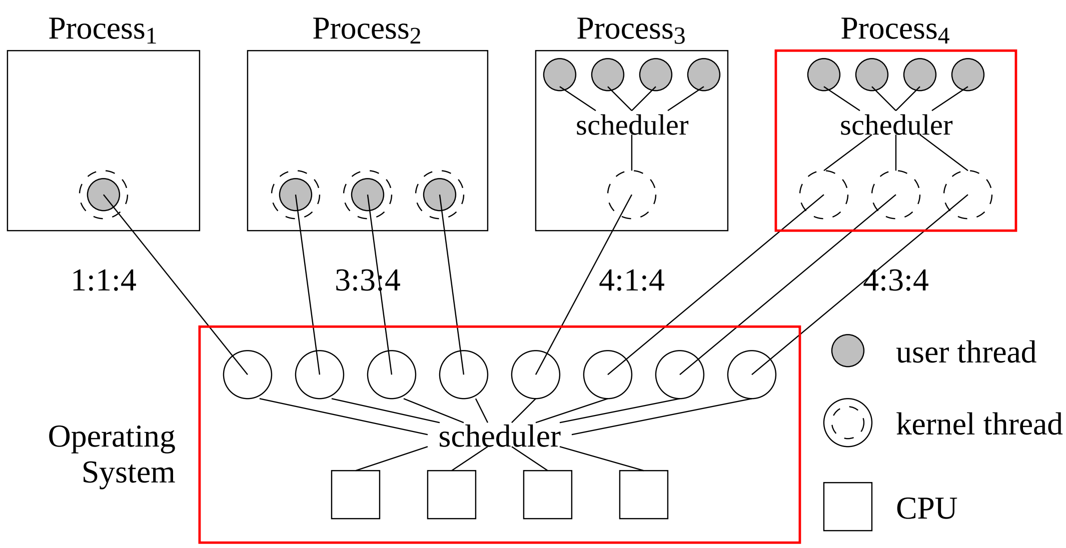
- If both tasks increment 10 times, the expected result is 20.
- True for single instruction, false for load-store sequence.
- Many failure cases for load-store sequence where `i` does not reach 20.
- Remember, context switch saves and restores registers for each coroutine/task.

```
task0                         task1
1st iteration
ld   r1,i      (r1 <- 0)
add  r1,#1     (r1 <- 1)
                              1st iteration
                              ld   r1,i      (r1 <- 0)
                              add  r1,#1     (r1 <- 1)
                              st   r1,i      (i  <- 1)
                              2nd iteration
                              ld   r1,i      (r1 <- 1)
                              add  r1,#1     (r1 <- 2)
                              st   r1,i      (i  <- 2)
                              3rd iteration
                              ld   r1,i      (r1 <- 2)
                              add  r1,#1     (r1 <- 3)
                              st   r1,i      (i  <- 3)
1st iteration
st   r1,i      (i  <- 1)
```

- The 3 iterations of task1 are lost when overwritten by task0.
- Hence, sequential operations, however small (increment), are unsafe in a concurrent program.

5.5 Threading Model

- For multiprocessor systems, a **threading model** defines relationship between threads and CPUs.
- OS manages CPUs providing logical access via **kernel threads** (**virtual processors**) *scheduled* across the CPUs.



- More kernel threads than CPUs to provide multiprocessing, i.e., run multiple programs simultaneously.

- A process may have multiple kernel threads to provide parallelism if multiple CPUs.
- A program may have user threads scheduled on its process's kernel threads.
- User threads are a low-cost structuring mechanism, like routines, objects, coroutines (versus high-cost kernel thread).
- Relationship is denoted by user:kernel:CPU, where:
  - 1:1:C (kernel threading) – 1 user thread maps to 1 kernel thread
  - M:M:C (generalize kernel threading) – M $\times$ 1:1 kernel threads (Java/Pthreads)
  - N:1:C (user threading) – N user threads map to 1 kernel thread (no parallelism)
  - N:M:C (user threading) – N user threads map to M kernel threads (μC++)
- Often the CPU number (C) is omitted.
- Can recursively add **nano threads** on top of user threads, and **virtual machine** below OS.

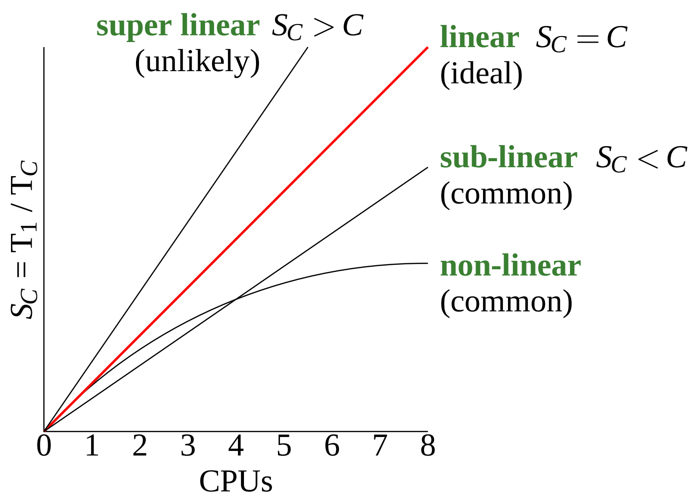
5.6 Concurrent Systems

- 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 type 1, there is a fundamental limit to how much parallelism can be found and current techniques only work on certain kinds of programs.
- In type 2, concurrency is accessed indirectly via specialized mechanisms (e.g., pragmas or parallel **for**) and implicitly managed.
- In type 3, concurrency is directly accessed and explicitly managed.
- Cases 1 & 2 are always built from type 3.
- To solve all concurrency problems, threads need to be explicit.
- Both implicit and explicit mechanisms are complementary, and hence, can appear together in a single programming language.
- However, the limitations of implicit mechanisms require that explicit mechanisms always be available to achieve maximum concurrency.
- μC++ only has explicit mechanisms, but its design does not preclude 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.
- In all cases, as concurrency increases, so does the complexity to express and manage it.

5.7 Speedup

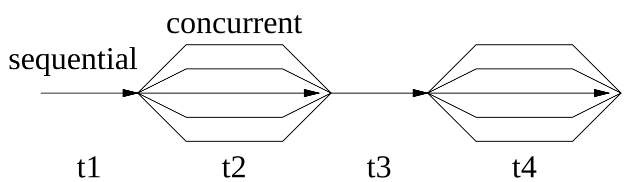
- Program **speedup** is $S_C = T_1/T_C$, where C is number of CPUs and T_1 is sequential execution.
- E.g., 1 CPU takes 10 seconds, $T_1 = 10$, 4 CPUs takes 2 seconds, $T_4 = 2 \Rightarrow S_4 = 10/2 = 5$ times speedup (linear).



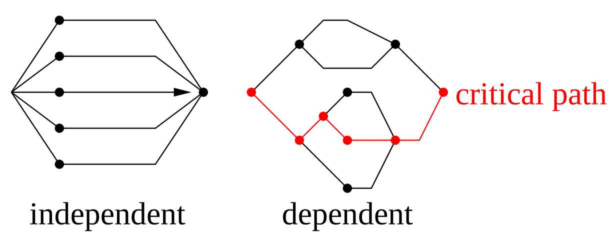
- Aspects affecting speedup (assume sufficient parallelism for concurrency):
  1. amount of concurrency
  2. critical path among concurrency
  3. scheduler efficiency
- An algorithm/program is composed of sequential and concurrent sections.
- E.g., sequentially read matrix, concurrently subtotal rows,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

$$S_C = \frac{1}{(1-P)+P/C} \quad where\ T_1 = 1, T_C = sequential + concurrent$$

- As C goes to infinity, P/C goes to 0, so maximum speedup is $1/(1-P)$, i.e., time for sequential section.
- Speedup falls rapidly as sequential section $(1-P)$ increases, especially for large C.
- E.g., sequential section = 0.1 (10%), $S_C = 1/(1-.9) \Rightarrow$ max speedup 10.
- Concurrent programming consists of minimizing sequential component $(1-P)$.
- E.g., an algorithm/program has 4 stages: $t1$ = 10, $t2$ = 25, $t3$ = 15, $t4$ = 50 (time units)
- Concurrently speedup sections $t2$ by 5 times and $t4$ by 10 times.



- T_C = 10 + 25 / 5 + 15 + 50 / 10 = 35 (time units)
  Speedup = 100 / 35 = 2.86 times
- Large reductions for $t2$ and $t4$ have only minor effect on speedup.
- Formula does not consider any increasing costs for the concurrency, i.e., administrative costs, so results are optimistic.
- While sequential sections bound speedup, concurrent sections bound speedup by the **critical path** of computation.



  - **independent execution** : all threads created together and do not interact.
  - **dependent execution** : threads created at different times and interact.
- Longest path bounds speedup.
- Finally, speedup can be affected by scheduler efficiency/ordering (often no control), e.g.:
  - greedy scheduling : run a thread as long as possible before context switching (not very concurrent).
  - LIFO scheduling : give priority to newly waiting tasks (starvation).
- Therefore, it is difficult to achieve significant speedup for many algorithms/programs.
- In general, benefit comes when many programs achieve some speedup so there is an overall improvement on a multiprocessor computer.

5.8 Concurrency

- Concurrency requires 3 mechanisms in a programming language.

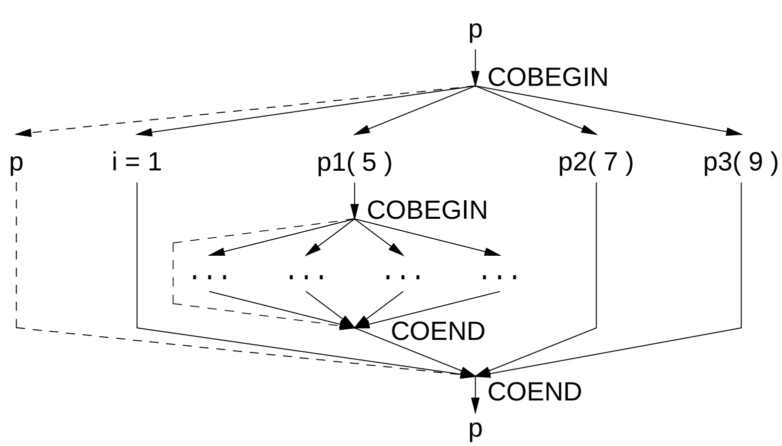
    1. creation – cause another thread of control to come into existence.
    2. synchronization – establish timing relationships among threads, e.g., same time, same rate, happens before/after.
    3. communication – transmit data among threads.

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

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

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

- A **thread graph** represents thread creations:



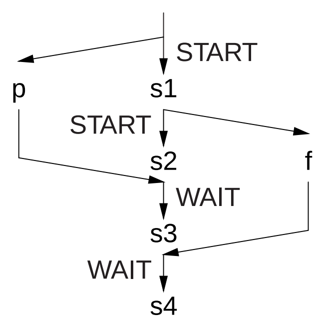
- Restricted to creating trees (lattice) of threads.

- Use recursion to create dynamic number of threads.

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

- What does the thread graph look like?
- Alternative approach for thread creation is START/WAIT, allowing arbitrary thread graph:

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



- COBEGIN/COEND can only approximate this thread graph:

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

- START/WAIT can simulate COBEGIN/COEND:

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

5.9 Thread Object

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

```
              _Task T {          // thread type
                  void main() {...} // thread starts here
              };
COBEGIN       {
                  T t;           // create object on stack, start thread
COEND         }                  // wait for thread to finish

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

- Block-terminate/delete must wait for each task's thread to finish. Why?
- Unusual to:
  - create objects in a block and not use it
  - allocate object and immediately delete it.
- Simulate COBEGIN/COEND with **_Task** object by creating type for each statement:

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

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

```
int i;                                  void uMain::main() {
_Task T1 {                                  T1 *tp = new T1;    // start T1
    void main() { p(5); }                   ... s1 ...
};                                          T2 *tf = new T2;    // start T2
_Task T2 {                                  ... s2 ...
    int temp;                               delete tp;          // wait for p
    void main() { temp = f(8); }            ... s3 ...
  public:                                   delete tf;          // wait for f
    ~T2() { i = temp; }                     ... s4 ...
};                                      }
```

- Variable i cannot be assigned until tf is deleted, otherwise the value could change in s2/s3.
- Allows same routine to be started multiple times with different arguments.

5.10 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 μC++)
  - because the parent terminates (not supported in μC++)
- Children can continue to exist even after the parent terminates (although this is rare).
  - E.g. sign off and leave child process(es) running
- Synchronizing at termination is possible for independent threads.
- Termination synchronization may be used to perform a final communication.

5.11 Divide-and-Conquer

- Divide-and-conquer is characterized by ability to subdivide work across data $\Rightarrow$ work can be performed independently on the data.
- Work performed on each data group is identical to work performed on data as whole.
- Taken to extreme, each data item is processed independently, but administration of concurrency becomes greater than cost of work.
- Only termination synchronization is required to know when the work is done
- Partial results are then processed further if necessary.
- Sum rows of a matrix concurrently using concurrent statement:

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

	matrix				subtotals
$T_0 \sum$	23	10	5	7	0
$T_1 \sum$	-1	6	11	20	0
$T_2 \sum$	56	-13	6	0	0
$T_3 \sum$	-2	8	-5	1	0
				total	$\sum$

- COFOR creates end - start threads, named start..end - 1, each executing loop body.
- Sum rows of a matrix concurrently using concurrent objects:

```
_Task Adder {
    int *row, cols, &subtotal; // communication
    void main() {
        subtotal = 0;
        for ( int c = 0; c < cols; c += 1 ) {
            subtotal += row[c];
        }
    }
  public:
    Adder( int row[ ], int cols, int &subtotal ) :
        row( row ), cols( cols ), subtotal( subtotal ) {}
};
void uMain::main() {
    const int rows = 10, cols = 10;
    int matrix[rows][cols], subtotals[rows], total = 0, r;
    // read matrix
    Adder *adders[rows];
    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];  // total subtotals
    }
    cout << total << endl;
}
```

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

5.12 Synchronization and Communication During Execution

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

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

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

```
_Task Cons {
    int N;
    void main() {
        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 ) {}
};
void uMain::main() {
    Prod prod( 5 ); Cons cons( 5 );
}
```

- 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.
- A loop waiting for an event among threads is called a **busy wait**.
- Are 2 synchronization flags necessary?

5.13 Communication

- Once threads are synchronized there are many ways that information can be transferred from one thread to the other.
- If the threads are in the same memory, then information can be transferred by value or address (e.g., reference parameter).
- If the threads are not in the same memory (distributed), then transferring information by value is straightforward but by address is difficult.

5.14 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 coroutine/task has its own stack (execution state)

- Nonlocal exceptions between a task and a coroutine are the same as between coroutines (single thread).
- Concurrent exceptions among tasks are more complex due to the multiple threads.
- A concurrent exception provides an additional kind of communication among tasks.
- For example, two tasks may begin searching for a key in different sets:

```
_Event StopEvent {};
_Task searcher {
    searcher &partner;
    void main() {
        try {
            . . .
            if ( key == . . . )
                _Resume 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.

```
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
}
```

5.15 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 threads 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, Amdahl's law) to maximize concurrency.**

5.16 Static Variables

- **Warning**: static variables in a class are shared among all objects generated by that class.
- These shared variables may need mutual exclusion for correct usage.
- However, a few special cases where **static** variables can be used safely, e.g., task constructor.
- If task objects are generated serially, **static** variables can be used in the constructor.
- E.g., assigning each task is own name:

```
_Task T {
    static int tid;
    string name; // must supply storage
    . . .
  public:
    T() {
        name = "T" + itostring( tid );  // shared read
        setName( name.c_str() );      // name task
        tid += 1;                      // shared write
    }
    . . .
};
int T::tid = 0;     // initialize static variable in .C file
T t[10];            // 10 tasks with individual names
```

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

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

- 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.

5.17 Mutual Exclusion Game

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

- Rules of the Game:

  1. Only one thread can be in a critical section at a time with respect to a particular object (**safety**).
  2. Threads may run at arbitrary speed and in arbitrary order, while the underlying system guarantees a thread makes progress (i.e., threads get some CPU time).
  3. If a thread is not in the entry or exit code controlling access to the critical section, it may not prevent other threads from entering the critical section.
  4. In selecting a thread for entry to a critical section, a selection cannot be postponed indefinitely (**liveness**). *Not* satisfying this rule is called **indefinite postponement** or **livelock**.
  5. After a thread starts entry to the critical section, it must eventually enter. *Not* satisfying this rule is called **starvation**.

- **Indefinite postponement and starvation are related by busy waiting.**

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

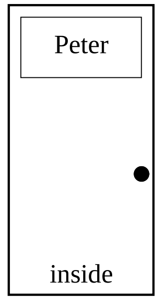
- Threads waiting to enter can be serviced in any order, as long as each thread eventually enters.

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

- If waiting threads are overtaken by a thread arriving later, there is the notion of **barging**.

5.18 Self-Testing Critical Section

```
uBaseTask *CurrTid;     // shared: current task id

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



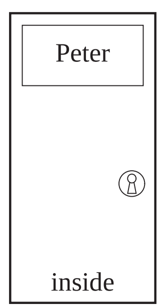
- What is the minimum number of interference tests and where?

- Why are multiple tests useful?

5.19 Software Solutions

5.19.1 Lock

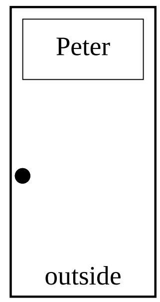
```
enum Yale {CLOSED, OPEN} Lock = OPEN; // shared

_Task PermissionLock {
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            while (::Lock == CLOSED) {} // entry protocol
            ::Lock = CLOSED;
            CriticalSection();      // critical section
            ::Lock = OPEN;          // exit protocol
        }
    }
  public:
    PermissionLock() {}
};
void uMain::main() {
    PermissionLock t0, t1;
}
```



Breaks rule 1

5.19.2 Alternation

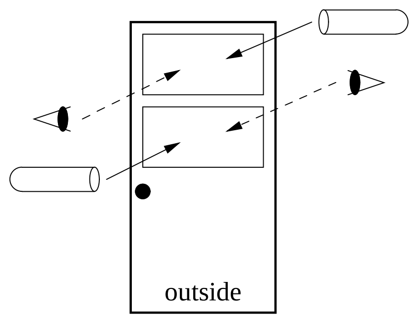
```
int Last = 0;                          // shared

_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

5.19.3 Declare Intent

```
enum Intent {WantIn, DontWantIn};

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



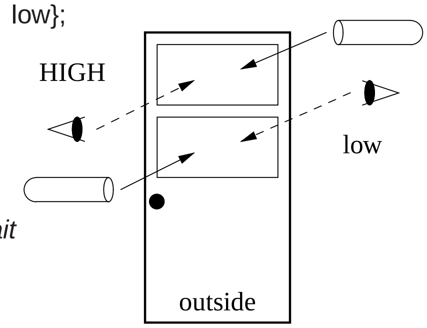
Breaks rule 4

5.19.4 Retract Intent

```
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

5.19.5 Prioritized Retract Intent

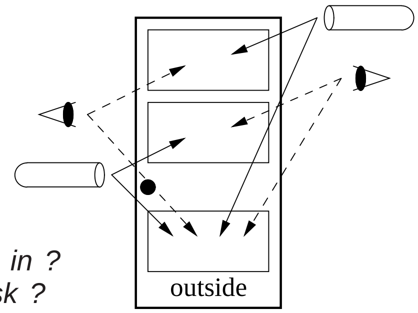



```
enum Intent {WantIn, DontWantIn}; enum Priority {HIGH, low};
_Task PriorityEntry {
    Intent &me, &you;  Priority priority;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            if ( priority == HIGH ) {    // entry protocol
                me = WantIn;             // high priority
                while ( you == WantIn ) {} // busy wait
            } else {                     // low priority
                for ( ;; ) {             // busy wait
                    me = WantIn;
                  if ( you == DontWantIn ) break;
                    me = DontWantIn;
                    while (you == WantIn) {} // busy wait
                }
            }
            CriticalSection();           // critical section
            me = DontWantIn;             // exit protocol
        }
    }
  public:
    PriorityEntry( Priority p, Intent &me, Intent &you ) : priority(p), me(me), you(you) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;
    PriorityEntry t0( HIGH, me, you ), t1( low, you, me );
} // main
```

Breaks rule 5

5.19.6 Dekker

- Structured version (Doran and Thomas, 1980)




```
enum Intent {WantIn, DontWantIn};
Intent *Last;
_Task Dekker {
    Intent &me, &you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            me = WantIn;            // declare intent
            if ( you == WantIn ) {  // other thread want in ?
                if ( ::Last == &me ) { // high priority task ?
                    me = DontWantIn; // retract intent
                    while ( ::Last == &me ) {}; // low priority busy wait
                    me = WantIn;    // re-declare intent
                }
                while ( you == WantIn ); // high priority busy wait
            }
            CriticalSection( id );
            ::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 );
}
```

- Alternative implementation nesting low priority loop in high priority loop.

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

- Dekker's algorithm is not **RW-safe**.
    - RW-safe means a mutual exclusion algorithm works if simultaneous writes scramble bits and simultaneous read/write reads flickering bits.

- ○ no simultaneous W/W scramble; fails for R/W flicker
- ○ RW-safe version exists

- Dekker has **unbounded overtaking** (not starvation) because *race loser retracts intent*.
- ⇒ thread exiting critical does not exclude itself for reentry.
  - ○ T0 exits critical section and attempts reentry
  - ○ T1 is now high priority (Last != me) but delays in low-priority busy-loop and resetting its intent.
  - ○ T0 can enter critical section unbounded times until T1 resets its intent
  - ○ T1 sets intent ⇒ bound of 1 as T1 can be entering or in critical section
- Unbounded overtaking is allowed by rule 3: not preventing entry to the critical section by the delayed thread.

5.19.7 Peterson

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

_Task Peterson {
    Intent &me, &you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
1           me = WantIn;           // entry protocol, order matters
2           ::Last = &me;          // RACE!
3           while ( you == WantIn && ::Last == &me ) {}
4           CriticalSection();     // critical section
5           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);
}
```

- Can line 2 be moved before 1?

```
1  2  ::Last = &me;         // RACE!
2  1  me = WantIn;          // entry protocol
3  3  while ( you == WantIn && ::Last == &me ) {}
4  4  CriticalSection();     // critical section
5  5  me = DontWantIn;      // exit protocol
```

  - ○ T0 executes Line 1 ⇒ ::Last = T0
  - ○ T1 executes Line 1 ⇒ ::Last = T1
  - ○ T1 executes Line 2 ⇒ T1 = WantIn

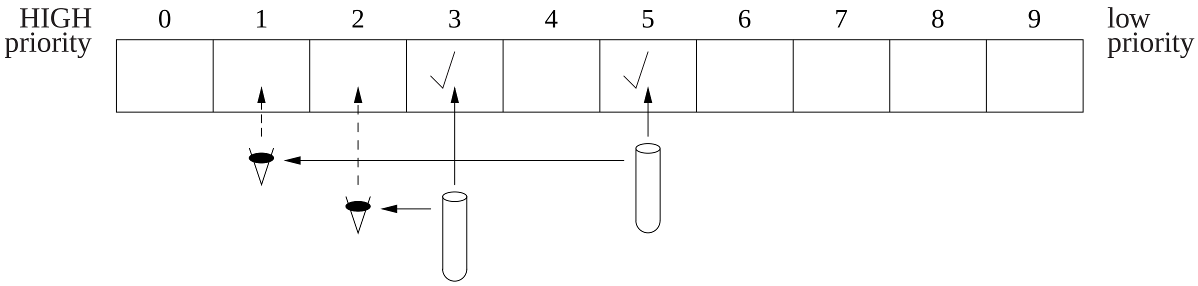
- ◦ T1 enters CS, because T0 == DontWantIn
- ◦ T0 executes Line 2 ⇒ T0 = WantIn
- ◦ T0 enters CS, because ::Last == T1

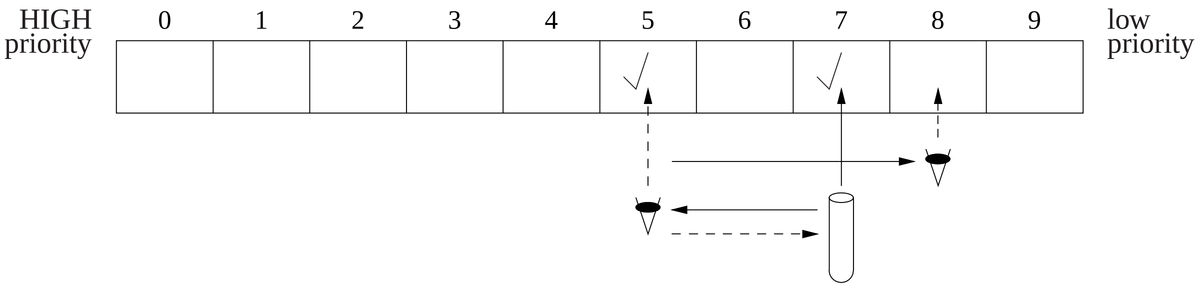
- Peterson's algorithm is RW-unsafe requiring atomic read/write operations.
- Peterson has **bounded overtaking** because *race loser does not retracts intent*.
- ⇒ thread exiting critical excludes itself for reentry.
  - T0 exits critical section and attempts reentry
  - T0 runs race by itself and loses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

5.19.8 N-Thread Prioritized Entry

```
enum Intent { WantIn, DontWantIn };
_Task NTask { // Burns/Lynch/Lamport: B-L
    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();
            intents[priority] = DontWantIn;   // exit protocol
        }
    }
  public:
    NTask( Intent i[], int N, int p ) : intents(i), N(N), priority(p) {}
};
```

Breaks rule 5





- Only N bits needed.
- No known solution for all 5 rules using only N bits.
- Other N-thread solutions use more memory.
  (best: 3-bit RW-unsafe, 4-bit RW-safe).

5.19.9 N-Thread Bakery (Tickets)

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

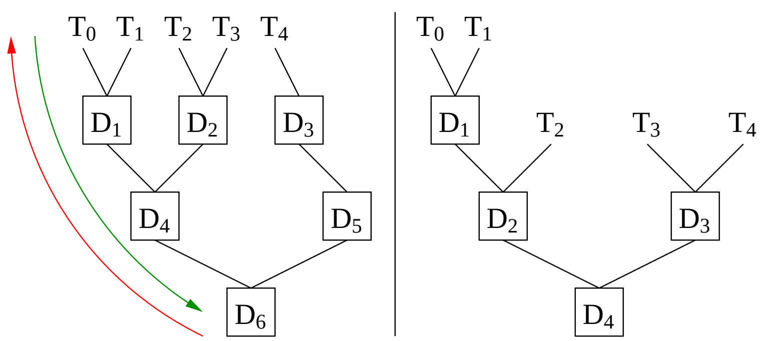
HIGH priority	0	1	2	3	4	5	6	7	8	9	low priority
	∞	∞	17	∞	∞	18	18	0	20	19	

- ticket value of ∞ (INT_MAX) $\Rightarrow$ don't want in
- ticket value of **0** $\Rightarrow$ selecting ticket
- ticket selection is unusual
- tickets are not unique $\Rightarrow$ use position as secondary priority
- low ticket and position $\Rightarrow$ high priority
- ticket values cannot increase indefinitely $\Rightarrow$ could fail (probabilistically correct)
- ticket value reset to INT_MAX when no attempted entry
- NM bits, where M is the ticket size (e.g., 32 bits)
- Lamport RW-safe
- Hehner/Shyamasundar RW-unsafe

  assignment ticket[priority] = max can flickers to INT_MAX $\Rightarrow$ other tasks proceed

5.19.10 Tournament

- Binary (d-ary) tree with $\lceil N/2 \rceil$ start nodes and $\lceil \lg N \rceil$ levels.



maximal **minimal**

- Thread assigned to start node, where it begins mutual exclusion process.
- Each node is like a Dekker or Peterson 2-thread algorithm, except exit protocol retracts intents in ***reverse*** order.
- Otherwise race between retracting/released threads along same tree path:
  - T_0 retracts its intent (left) at P_1,
  - T_1 (right) now moves to P_2 and set its intent in the shared variable (left),
  - T_0 continues and resets shared left intent at P_2,

- ◦ T_1 (left) now has wrong intent so T_2 thinks it does not want in.

- No overall livelock because each node has no livelock.

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

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

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

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

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

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

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

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

5.19.11 Arbiter

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

```
bool intent[N], serving[N];             // 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] ) {}  // busy wait
            CriticalSection();
            serving[id] = false;        // exit protocol
        }
    }
  public:
    Client( int me ) : me( me ) {}
};

_Task Arbiter {
    void main() {
        int i = N;                      // force cycle to start at id=0
        for ( ;; ) {
            do {                        // circular search => no starvation
                i = (i + 1) % N;        // advance next client
            } while ( ! intents[i] );     // not want in ?
            intents[i] = false;         // retract intent on behalf of client
            serving[i] = true;          // wait for exit from critical section
            while ( serving[i] ) {}     // busy wait
        }
    }
};
```

- Mutual exclusion becomes synchronization between arbiter and clients.
- Arbiter never uses the critical section ⇒ no indefinite postponement.
- Arbiter cycles through waiting clients (not FCFS) ⇒ no starvation.
- RW-unsafe due to read flicker.
- Cost is creation, management, and execution (continuous busy waiting) of arbiter task.

5.20 Hardware Solutions

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

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

  E.g., control order and speed of execution.
- Allows elimination of much of the shared information and the checking of this information required in the software solution.
- Special instructions to perform an **atomic read and write operation**.
- Sufficient for multitasking on a single CPU.
- Simple lock of critical section fails:

```
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.

```
while( Lock == CLOSED ? CLOSED : (Lock = CLOSED), OPEN );
// 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 guarantee of eventual progress.
- *Unfortunately, there is no such atomic construct in C.*
- Atomic hardware instructions can be used to achieve this effect.

5.20.1 Test/Set Instruction

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

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

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

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

5.20.2 Swap Instruction

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

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

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

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

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

5.20.3 Fetch and Increment

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

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

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

```
void Task::main() { // each task does
    while ( FetchInc( Lock ) != 0 );
    // critical section
    Lock = 0;
```

- Often fetch-and-increment is generalized to add any value ⇒ also decrement with negative value.
- Lock counter can overflow during busy waiting and starvation (rule 5).
- Use ticket counter to solve both problems (Bakery Algorithm, see Section 5.19.9, p. 78):

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

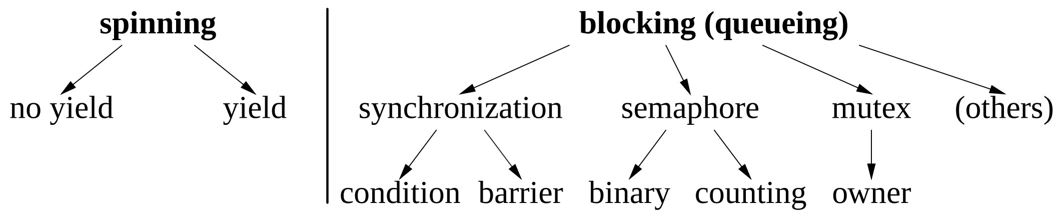
- Ticket overflow is a problem only if all values used simultaneously, and FIFO service ⇒ no starvation.

6 Lock Abstraction

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

6.1 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.

6.2 Spin Lock

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

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

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

  Hence, CPU is wasting time constantly checking the event.

- To increase uniprocessor efficiency, a task can:
  - explicitly terminate its time-slice
  - move back to the ready state after only ***one*** event-check fails. (Why one?)
- Task member yield relinquish time-slice by putting running task back onto ready queue.

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

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

- Most spin-lock implementations break rule 5, i.e., no bound on service.

  ⇒ possible starvation of one or more tasks.

- Spin lock is appropriate and necessary in situations where there is no other work to do.

6.2.1 Implementation

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

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

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

- Both locks are built directly from an atomic hardware instruction.

- Lock starts closed (0) or opened (1); waiting tasks compete to acquire lock after release.

- In theory, starvation could occur; in practice, it is seldom a problem.

- tryacquire makes one attempt to acquire the lock, i.e., it does not wait.

- 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

```
_Task T1 {
    uLock &lk;
    void main() {
        . . .
        S1
        lk.release();
        . . .
    }
  public:
    T1( uLock &lk ) : lk(lk) {}
};
```

```
_Task T2 {
    uLock &lk;
    void main() {
        . . .
        lk.acquire();
        S2
        . . .
    }
  public:
    T2( uLock &lk ) : lk(lk) {}
};
```

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

- mutual exclusion

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

```
void uMain::main() {
    uLock lock( 1 ); // open
    T t0( lock ), t1( lock );
}
```

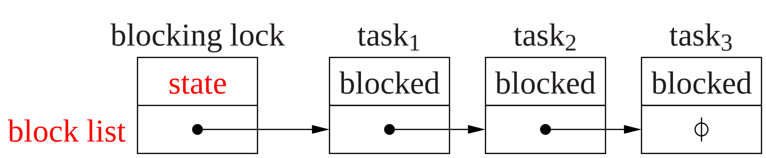
◦ Does this solution afford maximum concurrency?

◦ Depends on critical sections: **independent** (disjoint) or **dependent**.

◦ How many locks are needed for mutual exclusion?

6.3 Blocking Locks

- For spinning locks,
  - acquiring task(s) is solely responsible for detecting an open lock after the releasing task opens it.
- For blocking locks,
  - acquiring task makes ***one*** check for open lock and blocks
  - releasing task has sole responsibility for detecting blocked acquirer and transferring lock, or just releasing lock.
- Blocking locks reduce busy waiting by having releasing task do additional work: **cooperation**.
  - What advantage does the releasing task get from doing the cooperation?
- Therefore, all blocking locks have
  - state to facilitate lock semantics
  - list of blocked acquirers



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

6.3.1 Mutex Lock

- **Mutex lock** is used solely to provide mutual exclusion.
- Restricting a lock to just mutual exclusion:
  - separates lock usage between synchronization and mutual exclusion
  - permits optimizations and checks as the lock only provides one specialized function
- Mutex locks are divided into two kinds:
  - **single acquisition** : task that acquired the lock cannot acquire it again
  - **multiple acquisition** : lock owner can acquire it multiple times, called an **owner lock**
- Multiple acquisition can handle looping or recursion involving a lock:

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

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

6.3.1.1 Implementation

- Multiple acquisition lock manages owner state.

```
class MutexLock {
    queue<Task> blocked;        // blocked tasks
    SpinLock lock;              // mutex nonblocking lock
    bool inUse;                 // resource being used ?
    Task *owner                 // lock owner

  public:
    MutexLock() : inUse( false ), owner( NULL ) {}
    void acquire() {
        lock.acquire();         // barging
        while ( inUse && owner != currThread() ) { // busy waiting
            // add self to lock's blocked list
            yieldNoSchedule();
            lock.acquire();     // reacquire spinlock
        }
        inUse = true;
        owner = currThread();   // set new owner
        lock.release();
    }
```

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

- Single or multiple unblock for multiple acquisition?
- No cooperation: inUse flag is reset ⇒ acquiring tasks can barge ahead of released task.
- Released task must check again (**while**) ⇒ no progress guarantee (starvation).
- Blocking

```
// add self to blocked list of lock
lock.release();        // allow lock owner to release and unblock next waiting task
// PREEMPTION
yieldNoSchedule();     // yield CPU but do not reschedule onto ready queue
```

- Race between the blocking and unblocking tasks.
- Blocking task releases spin lock but preempted *before* yield and put onto ready queue.
- Unblocking task can enter, see blocking task on lock's blocked list, and put on ready queue.
- But task is already on the ready queue because of the preemption!
- Need *magic* to atomically yield without scheduling *and* release the spin lock.
- Magic is often accomplished with more cooperation:

```
yieldNoSchedule( lock );      // yield CPU but do not reschedule onto ready queue
```

- Spin lock is passed to the runtime system, which does the yield without schedule and then, on behalf of the user, unlocks the lock.
- Note, the runtime system violates order and speed of execution by being non-preemptable.
- Cooperation: hold inUse flag between releasing and unblocking task ⇒ bargers block so released task does not busy wait (**if**).

```
void acquire() {
    lock.acquire();             // barging
    if ( inUse && owner != currThread() ) { // NO barging
        // add self to lock's blocked list
        yieldNoSchedule( lock );
        lock.acquire();         // reacquire spinlock
    }
    inUse = true;
    owner = currThread();  // set new owner
    lock.release();
}

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

- Released task still waits to reacquire lock. (Is this necessary?)
- Cooperation: hold lock between releasing and unblocking task ⇒ NO bargers can enter.

```
void acquire() {
    lock.acquire();             // NO barging
    if ( inUse && owner != currThread() ) {
        // add self to lock's blocked list
        yieldNoSchedule( lock );
        // DO NOT REACQUIRE LOCK
    }
    inUse = true;
    owner = currThread();  // set new owner
    lock.release();
}

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

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

- Released task does not reacquire lock.
- ***critical section is not bracketed by the spin lock.***
- Cooperation: leave lock owner at front of blocked list as both flag and owner variable.

```
class MutexLock {
    queue<Task> blocked;        // blocked tasks
    SpinLock lock;              // nonblocking lock
  public:
    void acquire() {
        lock.acquire();         // barging
        if ( blocked.empty() ) { // no one waiting ?
            // add self to lock's blocked list
            lock.release();
        } else if ( blocked.head() == currThread() ) { // owner ?
            lock.release();
        } else {
            // add self to lock's blocked list
            yieldNoSchedule( lock );
            // DO NOT REACQUIRE LOCK
        }
    }

    void release() {
        lock.acquire();
        // REMOVE TASK FROM HEAD OF BLOCKED LIST
        if ( ! blocked.empty() ) {
            // MAKE TASK AT FRONT READY BUT DO NOT REMOVE
        }
        lock.release();         // NO RACE
    }
};
```

- If critical section acquired, blocked list must have a node on it check for in-use.
- Always release spin in release as unblocked task accesses no variables before exiting acquire.

6.3.1.2 uOwnerLock

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

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

- owner() returns NULL if no owner, otherwise address of task that currently owns lock.

- times() returns number of times lock has been acquired by owner task.
- Must release as many times as acquire.
- Otherwise, operations same as for uLock but with blocking instead of spinning for acquire.

6.3.1.3 Lock-Release Pattern

- To ensure a mutual exclusion lock is always released use the following patterns.
  - executable statement – finally clause

```
uOwnerLock lock;
lock.acquire();
try {
    . . .                    // protected by lock
} _Finally {
    lock.release();
}
```

  - allocation/deallocation (RAII – Resource Acquisition Is Initialization)

```
class RAII {                 // create once
    uOwnerLock &lock;
  public:
    RAII( uOwnerLock &lock ) : lock( lock ) { lock.acquire(); }
    ~RAII() { lock.release(); }
};
uOwnerLock lock;
{
    RAII raii( lock );       // lock acquired by constructor
    . . .                    // protected by lock
}                            // lock release by destructor
```

- Lock always released on normal, local transfer (**break**/**return**), and exception.

6.3.1.4 Stream Locks

- Specialized mutex lock for I/O based on uOwnerLock.
- Concurrent use of C++ streams can produce unpredictable results.
  - if two tasks execute:

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

  any of the outputs can appear:

abc def uvw xyz	abc uvw xyz def	uvw abc xyz def	abuvwc dexf yz	uvw abc def xyz

- μC++ provides: osacquire for output streams and isacquire for input streams.

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

```
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:

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

- Which locking pattern is used by stream locks?

6.3.2 Synchronization Lock

- **Synchronization lock** is used solely to block tasks waiting for synchronization.
- Weakest form of blocking lock as its only state is list of blocked tasks.
  - $\Rightarrow$ ***acquiring task always blocks*** (no state to make it conditional)
    Need ability to yield time-slice and block versus yield and go back on ready queue.
  - $\Rightarrow$ ***release is lost when no waiting task*** (no state to remember it)
- Often called a **condition lock**, with wait / signal(notify) for acquire / release.

6.3.2.1 Implementation

- Like mutex lock, synchronization lock needs mutual exclusion for safe implementation.
- Location of mutual exclusion classifies synchronization lock:

  external locking use an external lock to protect task list,

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

- external locking

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

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

- Usage pattern:
  - Cannot enter a restaurant if all tables are full.
  - Must acquire a lock to check for an empty table because state can change.
  - If no free table, block on waiting-list until a table becomes available.

```
// shared variables
    MutexLock m;                    // external mutex lock
    SyncLock s;                     // synchronization lock
    bool flag = false;              // indicate if event has occurred
// acquiring task
    m.acquire();        // mutual exclusion to examine state & possibly block
    if ( ! flag ) {     // event not occurred ?
        s.acquire();    // block for event
    }
// releasing task
    m.acquire();        // mutual exclusion to examine state
    flag = true;        // raise flag
    s.release();        // possibly unblock waiting task
    m.release();        // release mutual exclusion
```

- ***Problem: acquiring task blocked holding external mutual-exclusion lock!***

- Modify usage pattern:

```
// acquiring task
    m.acquire();        // mutual exclusion to examine state & possibly block
    if ( ! flag ) {     // event not occurred ?
        m.release(); // release external mutex-lock
        CAN BE INTERRUPTED HERE
        s.acquire();    // block for event
    }
```

- ***Problem: race releasing mutual-exclusion lock and blocking on synchronization lock.***
- To prevent race, modify synchronization-lock acquire to release lock.

```
void acquire( MutexLock &m ) {
    // add self to task list
    yieldNoSchedule( m );
}
```

- Or, protecting mutex-lock is bound at synchronization-lock creation and used implicitly.
- Now use first usage pattern.

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

- Has the race been prevented?
- Yes, because of the magic in yieldNoSchedule, which blocks and releases the mutex lock atomically.
- internal locking

```
class SyncLock {
    Task *list;          // blocked tasks
    SpinLock lock;       // internal lock
  public:
    SyncLock() : list( NULL ) {}
    void acquire( MutexLock &m ) {
        lock.acquire();
        // add self to task list
        m.release();     // release external mutex-lock
        CAN BE INTERRUPTED HERE
        yieldNoSchedule( lock );
        m.acquire();     // possibly reacquire after blocking
    }

    void release() {
        lock.acquire();
        if ( list != NULL ) {
            // remove task from blocked list and make ready
        }
        lock.release();
    }
};
```

  - Why does acquire still take an external lock?
  - Why is the race after releasing the external mutex-lock not a problem?
- Has the busy wait been removed from the blocking lock?

6.3.2.2 uCondLock

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

```
class uCondLock {
  public:
    uCondLock();
    bool 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 signal, except all waiting tasks are unblocked.

6.3.2.3 Programming Pattern

- Using synchronization locks is complex because they are weak.

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

- Why is synchronization more complex for blocking locks than spinning (uLock)?

```
bool done = false;

_Task T1 {
    uOwnerLock &mlk;
    uCondLock &clk;

    void main() {
        mlk.acquire();  // prevent lost signal
        if ( ! done )    // signal occurred ?
            // signal not occurred
            clk.wait( mlk ); // atomic wait/release
            // mutex lock re-acquired after wait
        mlk.release();  // release either way
        S2;
    }
  public:
    T1( uOwnerLock &mlk,
        uCondLock &clk ) :
        mlk(mlk), clk(clk) {}
};
```

```
_Task T2 {
    uOwnerLock &mlk;
    uCondLock &clk;

    void main() {
        S1;
        mlk.acquire();  // prevent lost signal
        done = true;    // remember signal occurred
        clk.signal();   // signal lost if not waiting
        mlk.release();
    }
  public:
    T2( uOwnerLock &mlk,
        uCondLock &clk ) :
        mlk(mlk), clk(clk) {}
};
```

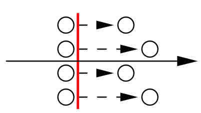
```
void uMain::main() {
    uOwnerLock mlk;
    uCondLock clk;
    T1 t1( mlk, clk );
    T2 t2( mlk, clk );
}
```

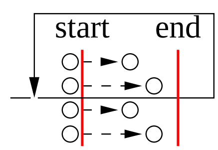
6.3.3 Barrier

- A **barrier** coordinates a group of tasks performing a concurrent operation surrounded by sequential operations.
- Hence, a barrier is for synchronization and cannot build mutual exclusion.
- Unlike previous synchronization locks, a barrier retains state about the events it manages: number of tasks blocked on the barrier.
- Since manipulation of this state requires mutual exclusion, most barriers use internal locking.
- E.g., 3 tasks must execute a section of code in a particular order: S1, S2 and S3 must *all* execute before S5, S6 and S7.

```
T1::main() {        T2::main() {        T3::main() {
    ...                 ...                 ...
    S1                  S2                  S3
    b.block();          b.block();          b.block();
    S5                  S6                  S7
    ...                 ...                 ...
}                   }                   }

void uMain::main() {
    Barrier b( 3 );
    T1 x( b );
    T2 y( b );
    T3 z( b );
}
```

- Barrier is initialized to control 3 tasks and passed to each task by reference (not copied).
- Barrier blocks each task at call to block until all tasks have called block.
- Last task to call block does not block and releases other tasks (cooperation).
- Hence, all tasks leave together (synchronized) after arriving at the barrier.
- Note, must specify in advance total number of block operations before tasks released.
- Why not use termination synchronization and create new tasks for each computation?
    - creation and deletion of computation tasks is expensive
- Two common uses for barriers:





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

6.3.3.1 uBarrier

- μC++ barrier is a thread-safe coroutine, where the coroutine main can be resumed by the last task arriving at the barrier.

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

- uBarrier has implicit mutual exclusion ⇒ no barging ⇒ only manage synchronization
- User barrier is built by:
  - inheriting from uBarrier
  - redefining block member and possibly coroutine main
  - possibly initializing main from constructor
- E.g., previous matrix sum (see page 66), using termination synchronization, adds subtotals in order of task termination, but barrier can add subtotals in order produced.

```
_Cormonitor Accumulator : public uBarrier {
    int total_;
    uBaseTask *Nth_;
    void main() {                     // resumed by last()
        Nth_ = &uThisTask();          // remember Nth task
        uBarrier::main();             // restart Nth task
    }
  public:
    Accumulator( int rows ) : uBarrier( rows ), total_( 0 ), Nth_( 0 ) {}
    void block( int subtotal ) { total_ += subtotal; uBarrier::block(); }
    int total() { return total_; }
    uBaseTask *Nth() { return Nth_; }
};

_Task Adder {
    int *row, size;
    Accumulator &acc;
    void main() {
        int subtotal = 0;
        for ( unsigned int r = 0; r < size; r += 1 ) subtotal += row[r];
        acc.block( subtotal );         // provide subtotal; wait for completion
    }
  public:
    Adder( int row[], int size, Accumulator &acc ) :
        size( size ), row( row ), acc( acc ) {}
};
```

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

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

6.3.4 Binary Semaphore

- **Binary semaphore** (Edsger W. Dijkstra) is blocking equivalent to yielding spin-lock.
- Provides synchronization *and* mutual exclusion.

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

- More powerful than synchronization lock as it remembers state about an event, but cannot be conjoined with mutex lock.
- 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 waits if the semaphore counter is zero and then decrements it.
- 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 has only only two states (open/closed), it is called a **binary semaphore**.

- synchronization

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

```
_Task T2 {
    BinSem &lk;
    void main() {
        . . .
        lk.P();
        S2
        . . .
    }
  public:
    T2( BinSem &lk ) : lk(lk) {}
};
```

```
void uMain::main() {
    BinSem lock( 0 ); // closed
    T1 t1( lock );
    T2 t2( lock );
}
```

- mutual exclusion

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

```
void uMain::main() {
    BinSem lock( 1 ); // start open
    T t0( lock ), t1( lock );
}
```

6.3.4.1 Implementation

- Implementation has:
  - blocking task-list
  - cnt indicates if event has occurred (state)
  - spin lock to protect state

```
class BinSem {
        queue<Task> blocked;            // blocked tasks
        SpinLock lock;                  // mutex nonblocking lock
        bool inUse;                     // resource being used ?
    public:
        BinSem( bool start = 1 ) : inUse( start ) {}
        void P() {
            lock.acquire();
            if ( inUse ) {
                // add self to lock's blocked list
                yieldNoSchedule( lock );
                // DO NOT REACQUIRE LOCK
            }
            inUse = true;
            lock.release();
        }

        void V() {
            lock.acquire();
            if ( ! blocked.empty() ) {
                // remove task from blocked list and make ready
                // DO NOT RELEASE LOCK
            } else {
                inUse = false
                lock.release();         // NO RACE
            }
        }
};
```

- Same as mutexLock (except for owner check) but can set starting value of inUse.
- Higher cost for synchronization if external lock already acquired.

6.3.5 Counting Semaphore

- Augment the definition of P and V to allow a multi-valued semaphore.
- What does it mean for a lock to have more than open/closed (unlocked/locked)?
  - $\Rightarrow$ critical sections allowing N simultaneous tasks.
- Augment V to allow increasing the counter an arbitrary amount.
- synchronization
  - Three tasks must execute so S2 and S3 only execute after S1 has completed.

```
T1::main() {          T2::main() {          T3::main() {
    ...                   ...                   S1
    lk.P();               lk.P();               lk.V(); // lk.V(2)
    S2                    S3                    lk.V();
    ...                   ...                   ...
}                     }                     }
void uMain::main() {
    CntSem lock( 0 ); // closed
    T1 x( lock );
    T2 y( lock );
    T3 z( lock );
}
```

- mutual exclusion
  - Critical section allowing up to 3 simultaneous tasks.

```
_Task T {                             void uMain::main() {
    CntSem &lk;                           CntSem lock( 3 ); // allow 3
    void main() {                         T t0( lock ), t1( lock ), ...;
        ...                           }
        lk.P();
        // up to 3 tasks in
        // critical section
        lk.V();
        ...
    }
  public:
    T( BinSem &lk ) : lk(lk) {}
};
```

- Must know in advance the total number of P's on the semaphore.

6.3.5.1 Implementation

- Change flag into counter, and set to some maximum on creation.
- Decrement counter on acquire and increment on release.
- Block acquiring task when counter is 0.
- Negative counter indicates number of waiting tasks.

```
class CntSem {
    queue<Task> blocked;    // blocked tasks
    int cnt;                // resource being used ?
    SpinLock lock;          // nonblocking lock
  public:
    CntSem( int start = 1 ) : cnt( start ) {}
    void P() {
        lock.acquire();
        cnt -= 1;
        if ( cnt < 0 ) {
            // add self to lock's blocked list
            // magically yield, block and release spin lock
            // UNBLOCK WITH SPIN LOCK ACQUIRED
        }
        lock.release();
    }

    void V() {
        lock.acquire();
        cnt += 1;
        if ( cnt <= 0 ) {
            // remove task from blocked list and make ready
            // CANNOT ACCESS ANY STATE
        } else {
            lock.release();    // conditionally release lock
        }
    }
};
```

- In general, binary/counting semaphores are used in two distinct ways:

  1. For synchronization, if the semaphore starts at 0 $\Rightarrow$ waiting for an event to occur.
  2. For mutual exclusion, if the semaphore starts at 1(N) $\Rightarrow$ controls a critical section.

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

```
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 Ved 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.

6.4 Lock Programming

6.4.1 Precedence Graph

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

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

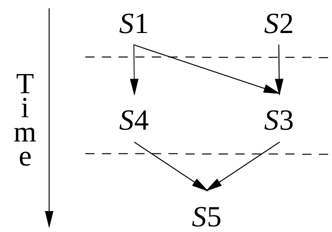
```
S1: a := 1
S2: b := 2
S3: c := a + b
S4: d := 2 * a
S5: e := c + d
```

- Analyse which data and code depend on each other.

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

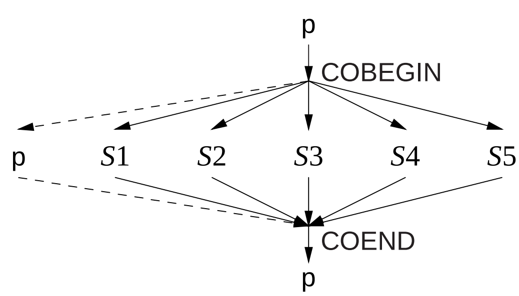
- Statement S3 is dependent on S1 and S2 because it uses both results.

- Display dependences graphically in a **precedence graph** (different from process graph).



```
Semaphore L1(0), L2(0), L3(0), L4(0);
COBEGIN
    BEGIN a := 1; V(L1); END;
    BEGIN b := 2; V(L2); END;
    BEGIN P(L1); P(L2); c := a + b; V(L3); END;
    BEGIN P(L1); d := 2 * a; V(L4); END;
    BEGIN P(L3); P(L4); e := c + d; END;
COEND
```

- Does this solution work?
- process graph (different from precedence graph)

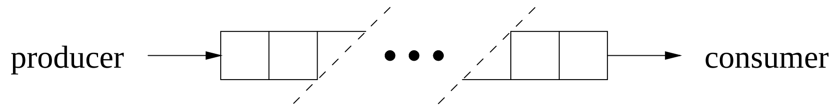


6.4.2 Buffering

- Tasks communicate unidirectionally through a queue.
- Producer adds items to the back of a queue.
- Consumer removes items from the front of a queue.

6.4.2.1 Unbounded Buffer

- Two tasks communicate through a queue of unbounded length.



- Because tasks work at different speeds, producer may get ahead of consumer.
  - Producer never has to wait as buffer has infinite length.
  - Consumer has to wait if buffer is empty ⇒ wait for producer to add.
- Queue is shared between producer/consumer, and counting semaphore controls access.

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

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

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

6.4.2.2 Bounded Buffer

- Two tasks communicate through a queue of bounded length.

- Because of bounded length:

  - Producer has to wait if buffer is full ⇒ wait for consumer to remove.
  - Consumer has to wait if buffer is empty ⇒ wait for producer to add.

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

```
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?

34	13	9	10	-3

full	empty
$\cancel{0}$	$\cancel{5}$
$\cancel{1}$	$\cancel{4}$
$\cancel{2}$	$\cancel{3}$
$\cancel{3}$	$\cancel{2}$
$\cancel{4}$	$\cancel{1}$
5	0

6.4.3 Lock Techniques

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

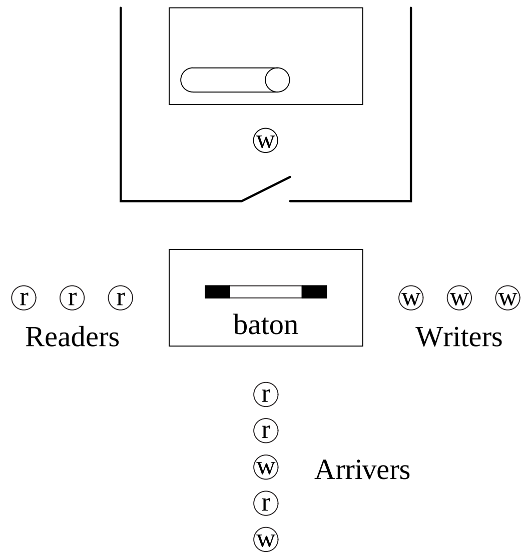
- Split binary semaphores can be used to solve complicated mutual-exclusion problems by a technique called **baton passing**.
- The rules of baton passing are:
  - there is exactly one (conceptual) baton
  - nobody moves in the entry/exit code unless they have it
  - once the baton is released, cannot read/write variables in entry/exit
- E.g., baton is conceptually acquired in entry/exit protocol and passed from signaller to signalled task (see page 90).

```
class BinSem {
        queue<Task> blocked;
        bool inUse;
        SpinLock lock;
    public:
        BinSem( bool usage = false ) : inUse( usage ) {}
        void P() {
            lock.acquire(); PICKUP BATON, CAN ACCESS STATE
            if ( inUse ) {
                // add self to lock's blocked list
                PUT DOWN BATON, CANNOT ACCESS STATE
                // magically yield, block and release spin lock
                // UNBLOCK WITH SPIN LOCK ACQUIRED
                PASSED BATON, CAN ACCESS STATE
            }
            inUse = true;
            lock.release(); PUT DOWN BATON, CANNOT ACCESS STATE
        }

        void V() {
            lock.acquire(); PICKUP BATON, CAN ACCESS STATE
            if ( ! blocked.empty() ) {
                // remove task from blocked list and make ready
                PASS BATON, CANNOT ACCESS STATE
            } else {
                inUse = false;
                lock.release(); PUT DOWN BATON, CANNOT ACCESS STATE
            }
        }
};
```

- Can mutex/condition lock perform baton passing?
  - Not if signalled task must implicitly re-acquire the mutex lock before continuing.
  - $\Rightarrow$ signaller must release the mutex lock.
  - There is now a race between signalled and calling tasks, resulting in barging.

6.4.4 Readers and Writer Problem

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



6.4.4.1 Solution 1

```
uSemaphore entry_q(1), read_q(0), write_q(0); // split binary semaphores
int r_del = 0, w_del = 0, r_cnt = 0, w_cnt = 0; // auxiliary counters
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: reader only checks for writer in resource, never writers waiting to use it.
  - ⇒ continuous stream of readers (actually only 2 needed) prevent waiting writers from making progress (starvation).

6.4.4.2 Solution 2

- Give writers priority and make the readers wait.
  - Works most of the time because normally 80% readers and 20% writers.
- Change entry protocol for reader to the following:

```
entry_q.P();                            // entry protocol
if ( w_cnt > 0 || w_del > 0 ) {         // waiting writers?
    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:

```
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.

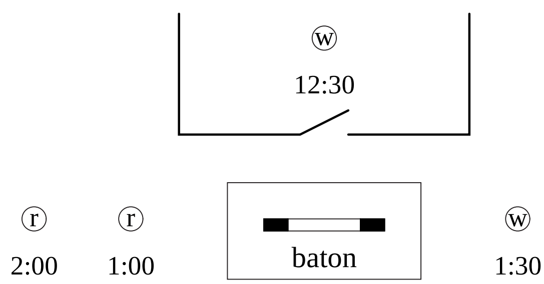
6.4.4.3 Solution 3

- Fairness on simultaneous arrival is solved by alternation (Dekker's solution).
- E.g., use "last" flag indicating the last kind of tasks to use resource, i.e., reader or writer.
- On exit, first select from opposite kind, e.g., if flag is reader, first check for waiting writer otherwise waiting reader, then update flag.
- Flag is unnecessary if readers wait when there is a waiting writer, and all readers started after a writer.
- ⇒ put writer's exit-protocol back to favour readers.

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

- Readers cannot barge ahead of waiting writers and a writer cannot barge ahead of a waiting reader ⇒ alternation for simultaneous waiting.
- There is still a problem: staleness/freshness.

6.4.4.4 Solution 4

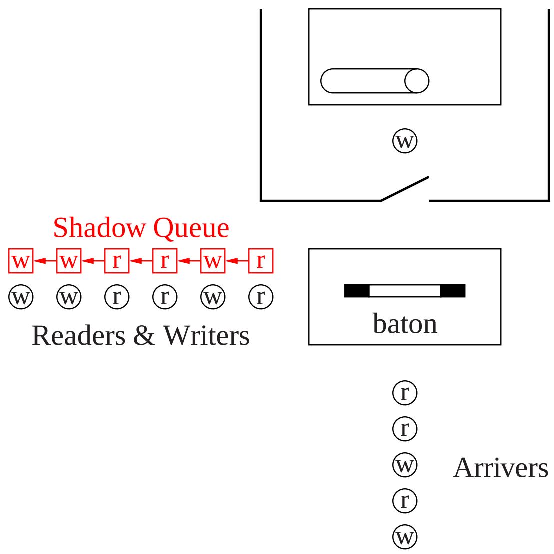


- Alternation for simultaneous waiting means when writer leaves resource either:
  - both readers enter ⇒ 2:00 reader reads data that is **stale**; should read 1:30 write
  - writer enters and overwrites 12:30 data (never seen) ⇒ 1:00 reader reads data that is too **fresh** (i.e., missed reading 12:30 data)
  - staleness/freshness can lead to plane or stock-market crash
- Service readers and writers in **temporal order**, i.e., first-in first-out (FIFO), but allow multiple concurrent readers.

- Implement by having readers and writers wait on same semaphore ⇒ collapse split binary semaphore.

- ***But now lose kind of waiting task!***

- Introduce shadow queue to retain kind of waiting task on semaphore:



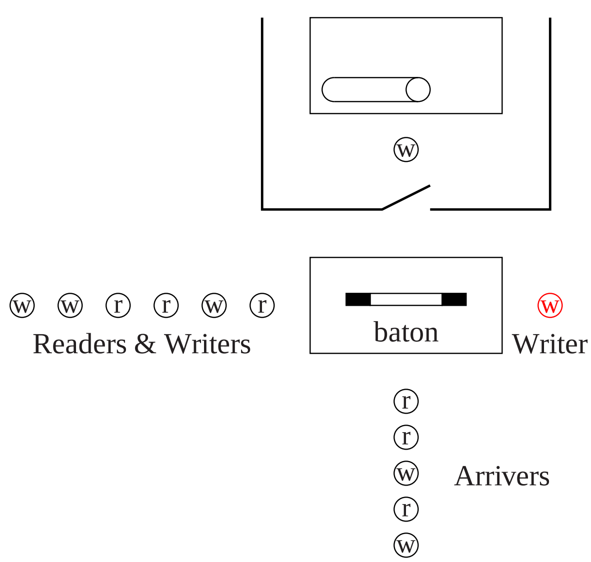

```
uSemaphore entry_q(1), rw_q(0);     // readers/writers, temporal order
int rw_del = 0, r_cnt = 0, w_cnt = 0; // auxiliary counters
enum RW { READER, WRITER };         // kinds of tasks
queue<RW> rw_id;                    // queue of kinds
void Reader::main() {
    entry_q.P();                    // entry protocol
    if ( w_cnt > 0 || rw_del > 0 ) { // anybody waiting?
        rw_id.push( READER );       // store kind
        rw_del += 1; entry_q.V(); rw_q.P();
        rw_id.pop();
    }
    r_cnt += 1;
    if ( rw_del > 0 && rw_id.front() == READER ) { // more readers ?
        rw_del -= 1; rw_q.V();      // pass baton
    } else
        entry_q.V();                // put baton down
    . . .
    entry_q.P();                    // exit protocol
    r_cnt -= 1;
    if ( r_cnt == 0 && rw_del > 0 ) { // last reader ?
        rw_del -= 1; rw_q.V();      // pass baton
    } else
        entry_q.V();                // put baton down
}
```

```
void Writer::main() {
    entry_q.P();                          // entry protocol
    if ( r_cnt > 0 || w_cnt > 0 ) {
        rw_id.push( WRITER );             // store kind
        rw_del += 1; entry_q.V(); rw_q.P();
        rw_id.pop();
    }
    w_cnt += 1;
    entry_q.V();
    . . .
    entry_q.P();                          // exit protocol
    w_cnt -= 1;
    if ( rw_del > 0 ) {                   // anyone waiting ?
        rw_del -= 1; rw_q.V();            // pass baton
    } else
        entry_q.V();                      // put baton down
}
```

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

6.4.4.5 Solution 5

- Cheat on cooperation:
    - allow 2 checks for write instead of 1
    - use reader/writer bench and writer chair.
- On exit, if chair empty, unconditionally unblock task at front of reader/writer semaphore.
- ⇒ ***reader can incorrectly unblock a writer.***
- This writer now waits second time but in chair.
- Chair is always checked first on exit (higher priority than bench).



```
uSemaphore entry_q(1), rw_q(0), write_q(0);
int rw_del = 0, w_del, r_cnt = 0, w_cnt = 0; // auxiliary counters
void Reader::main() {
    entry_q.P();                        // entry protocol
    if ( w_cnt > 0 || w_del > 0 || rw_del > 0 ) {
        rw_del += 1; entry_q.V(); rw_q.P();
    }
    r_cnt += 1;
    if ( rw_del > 0 ) {                 // more readers ?
        rw_del -= 1; rw_q.V();          // pass baton
    } else
        entry_q.V();                    // put baton down
    . . .
    entry_q.P();                        // exit protocol
    r_cnt -= 1;
    if ( r_cnt == 0 ) {                 // last reader ?
        if ( w_del != 0 ) {             // writer waiting ?
            w_del -= 1; write_q.V(); // pass baton
        } else if ( rw_del > 0 ) {      // anyone waiting ?
            rw_del -= 1; rw_q.V(); // pass baton
        } else {
            entry_q.V();                // put baton down
        }
    } else
        entry_q.V();                    // put baton down
}

void Writer::main() {
    entry_q.P();                        // entry protocol
    if ( r_cnt > 0 || w_cnt > 0 ) {
        rw_del += 1; entry_q.V(); rw_q.P();
        if ( r_cnt > 0 ) {              // wait once more ?
            w_del += 1; entry_q.V(); write_q.P();
        }
    }
    w_cnt += 1;
    entry_q.V();                        // put baton down
    . . .
    entry_q.P();                        // exit protocol
    w_cnt -= 1;
    if ( rw_del > 0 ) {                 // anyone waiting ?
        rw_del -= 1; rw_q.V();          // pass baton
    } else
        entry_q.V();                    // put baton down
}
```

6.4.4.6 Solution 6

- Still temporal problem when tasks move from one blocking list to another.
- In solutions, reader/writer entry-protocols have code sequence:

```
. . . entry_q.V(); INTERRUPTED HERE X_q.P();
```

- For writer:
  - pick up baton and see readers using resource
  - put baton down, entry.V(), but time-sliced before wait, X_q.P().
  - another writer does same thing, and this can occur to any depth.
  - writers restart in any order or immediately have another time-slice
  - e.g., 2:00 writer goes ahead of 1:00 writer ⇒ freshness problem.
- For reader:
  - pick up baton and see writer using resource
  - put baton down, entry.V(), but time-sliced before wait, X_q.P().
  - writers that arrived ahead of reader do same thing
  - reader restarts before any writers
  - e.g., 2:00 reader goes ahead of 1:00 writer ⇒ staleness problem.
- Need atomic block and release ⇒ magic like turning off time-slicing.

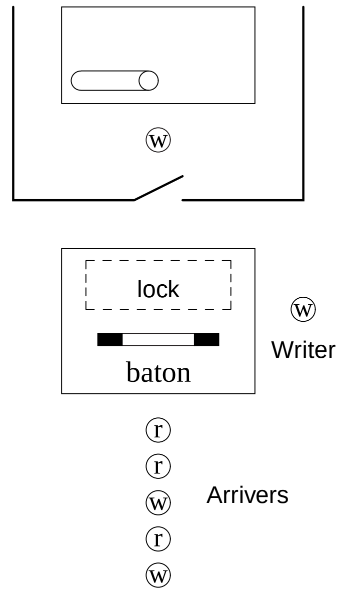
```
X_q.P( entry_q );    // uC++ semaphore
```

- Alternative: ticket
  - readers/writers take ticket (see Section 5.19.9, p. 78) before putting baton down
  - to pass baton, serving counter is incremented and then **WAKE ALL BLOCKED TASKS**
  - each task checks ticket with serving value, and one proceeds while others reblock
  - starvation not an issue as waiting queue is bounded length, but inefficient
- Alternative: private semaphore
  - list of **private semaphore**s, one for each waiting task, versus multiple waiting tasks on a semaphore.
  - add list node before releasing entry lock, which establishes position, then block on private semaphore.
  - to pass baton, private semaphore at head of the queue is Ved, if present.
  - if task blocked on private semaphore, it is unblocked
  - if task not blocked due to time-slice, V is remembered, and task does not block on P.

```
uSemaphore entry_q(1);
int r_cnt = 0, w_cnt = 0;
struct RWnode {
    RW rw;                          // kinds of task
    uSemaphore sem;                 // private semaphore
    RWnode( RW rw ) : rw(rw), sem(0) {}
};
queue<RWnode *> rw_id;
void Reader::main() {
    entry_q.P();                    // entry protocol
    if ( w_cnt > 0 || ! rw_id.empty() ) { // anybody waiting?
        RWnode r( READER );
        rw_id.push( &r );           // store kind
        rw_del += 1; entry_q.V(); r.sem.P();
        rw_id.pop();
    }
    r_cnt += 1;
    if ( rw_del > 0 && rw_id.front()->rw == READER ) { // more readers ?
        rw_del -= 1; rw_id.front()->sem.V(); // pass baton
    } else
        entry_q.V();                // put baton down
    . . .
    entry_q.P();                    // exit protocol
    r_cnt -= 1;
    if ( r_cnt == 0 && rw_del > 0 ) { // last reader ?
        rw_del -= 1; rw_id.front()->sem.V(); // pass baton
    } else
        entry_q.V();                // put baton down
}

void Writer::main() {
    entry_q.P();                    // entry protocol
    if ( r_cnt > 0 || w_cnt > 0 ) { // resource in use ?
        RWnode w( WRITER );
        rw_id.push( &w );           // remember kind of task
        rw_del += 1; entry_q.V(); w.sem.P();
        rw_id.pop();
    }
    w_cnt += 1;
    entry_q.V();
    . . .
    entry_q.P();                    // exit protocol
    w_cnt -= 1;
    if ( rw_del > 0 ) {             // anyone waiting ?
        rw_del -= 1; rw_id.front()->sem.V(); // pass baton
    } else
        entry_q.V();                // put baton down
}
```

6.4.4.7 Solution 7

- Ad hoc solution with questionable split-binary semaphores and baton-passing.



- Tasks wait in temporal order on entry semaphore.
- Only one writer ever waits on the writer chair until readers leave resource.
- ***Waiting writer blocks holding baton to force other arriving tasks to wait on entry.***
- Semaphore lock is used only for mutual exclusion.
- Sometimes acquire two locks to prevent tasks entering and leaving.
- Release in opposite order.

```
uSemaphore entry_q(1);              // two locks open
uSemaphore lock(1), writer_q(0);

void Reader::main() {
    entry_q.P();                    // entry protocol
    lock.P();
    r_cnt += 1;
    lock.V();
    entry_q.V();                    // put baton down
    . . .
    lock.P();                       // exit protocol
    r_cnt -= 1;                     // critical section
    if ( r_cnt == 0 && w_cnt == 1 ) { // last reader & writer waiting ?
        lock.V();
        writer_q.V();               // pass baton
    } else
        lock.V();
}
```

```
void Writer::main() {
    entry_q.P();                    // entry protocol
    lock.P();
    if ( r_cnt > 0 ) {              // readers waiting ?
        w_cnt += 1;
        lock.V();
        writer_q.P();               // wait for readers
        w_cnt -= 1;                 // unblock with baton
    } else
        lock.V();
    . . .
    entry_q.V();                    // exit protocol
}
```

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

7 Concurrent Errors

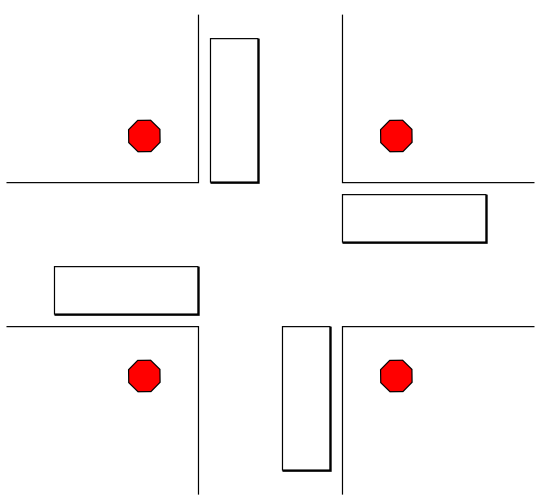
7.1 Race Condition

- A **race condition** occurs when there is missing:
  - synchronization
  - mutual exclusion
- Two or more tasks race along assuming synchronization or mutual exclusion has occurred.
- Can be very difficult to locate (thought experiments).
  - Aug. 14, 2003 Northeastern blackout : worst power outage in North American history.
  - Race condition buried in four million lines of C code.
  - “in excess of three million online operational hours in which nothing had ever exercised that bug.”

7.2 No Progress

7.2.1 Live-lock

- Indefinite postponement: “You go first” problem on simultaneous arrival (consuming CPU)
- Caused by poor scheduling in entry protocol:



- There always exists some mechanism to break tie on simultaneous arrival that deals effectively with live-lock.

7.2.2 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.
- Like live-lock, starving task might be ready at any time, switching among active, ready and possibly blocked states (consuming CPU).

7.2.3 Deadlock

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

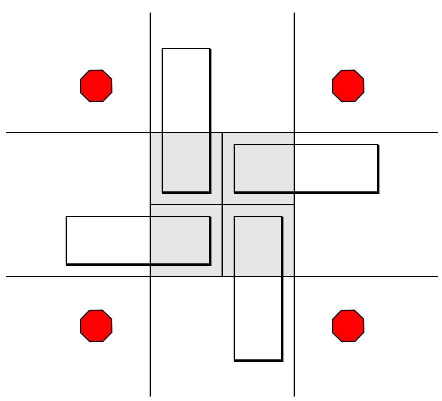
7.2.3.1 Synchronization Deadlock

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

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

7.2.3.2 Mutual Exclusion Deadlock

- Failure to acquire a resource protected by mutual exclusion.



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

- Simple example using semaphores:

```
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 more than one 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 runtime system cannot get it back (no preemption).
    4. There exists a circular wait of processes on resources.
    5. These conditions must occur simultaneously.

7.3 Deadlock Prevention

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

7.3.1 Synchronization Prevention

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

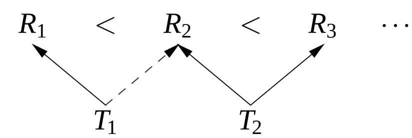
7.3.2 Mutual Exclusion Prevention

- 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:

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

- Use an **ordered resource** policy:



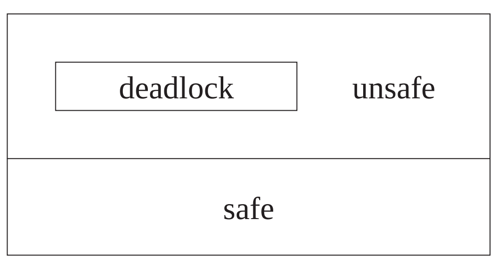
  - divide all resources into classes R_1, R_2, R_3, etc.
  - rule: can only request a resource from class R_i if holding no resources from any class R_j for $j \geq i$
  - unless each class contains only one resource, requires requesting several resources simultaneously
  - denote the highest class number for which T holds a resource by $h(T)$
  - if process T_1 is requesting a resource of class k and is blocked because that resource is held by process T_2, then $h(T_1) < k \leq h(T_2)$
  - as the preceding inequality is strict, a circular wait is impossible
  - 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.

7.4 Deadlock Avoidance

- Monitor all lock blocking and resource allocation to detect any potential formation of deadlock.



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

7.4.1 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.

	R1	R2	R3	R4	
T1	4	10	1	1	maximum needed
T2	2	4	1	2	for execution
T3	5	9	0	1	(M)
T1	$\cancel{2}3$	5	1	0	currently
T2	1	2	1	0	allocated
T3	1	2	0	0	(C)

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

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?

total available resources

R1	R2	R3	R4	
6	12	4	2	(TR)

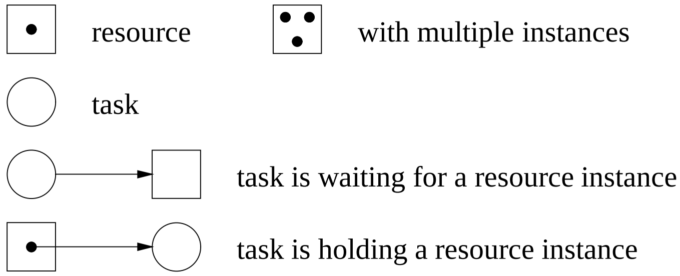
current available resources

	1	3	2	2	$(CR = TR - \sum C_{cols})$
T2	0	1	2	0	$(CR = CR - N_{T2})$
	2	5	3	2	$(CR = CR + M_{T2})$
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	$(CR = CR + M_{T3})$

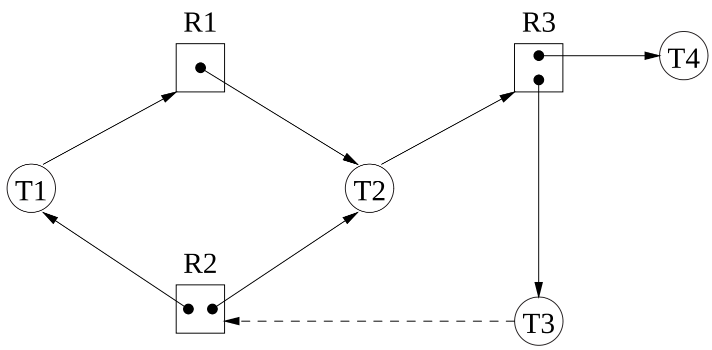
- If there is a choice of processes to choose for execution, it does not matter which path is taken.
- Example: If T1 or T3 could go to their maximum with the current resources, then choose either. A safe order starting with T1 exists if and only if a safe order starting with T3 exists.
- So a safe order 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.

7.4.2 Allocation Graphs

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



- Multiple instances are put into a resource so that a specific resource does not have to be requested. Instead, a generic request is made.

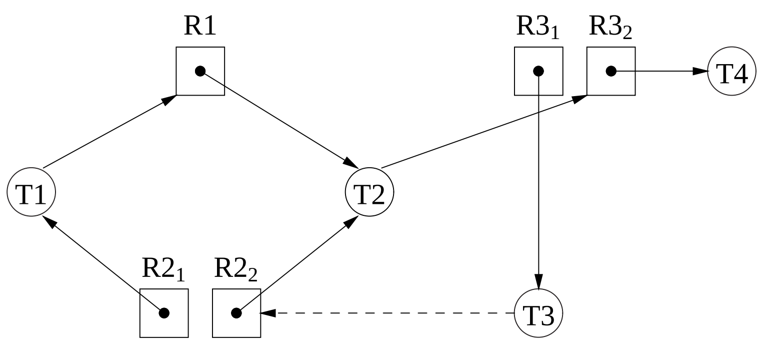


- If a graph contains no cycles, no process in the system is deadlocked.
- If any resource has several instances, a cycle $\not\Rightarrow$ deadlock.

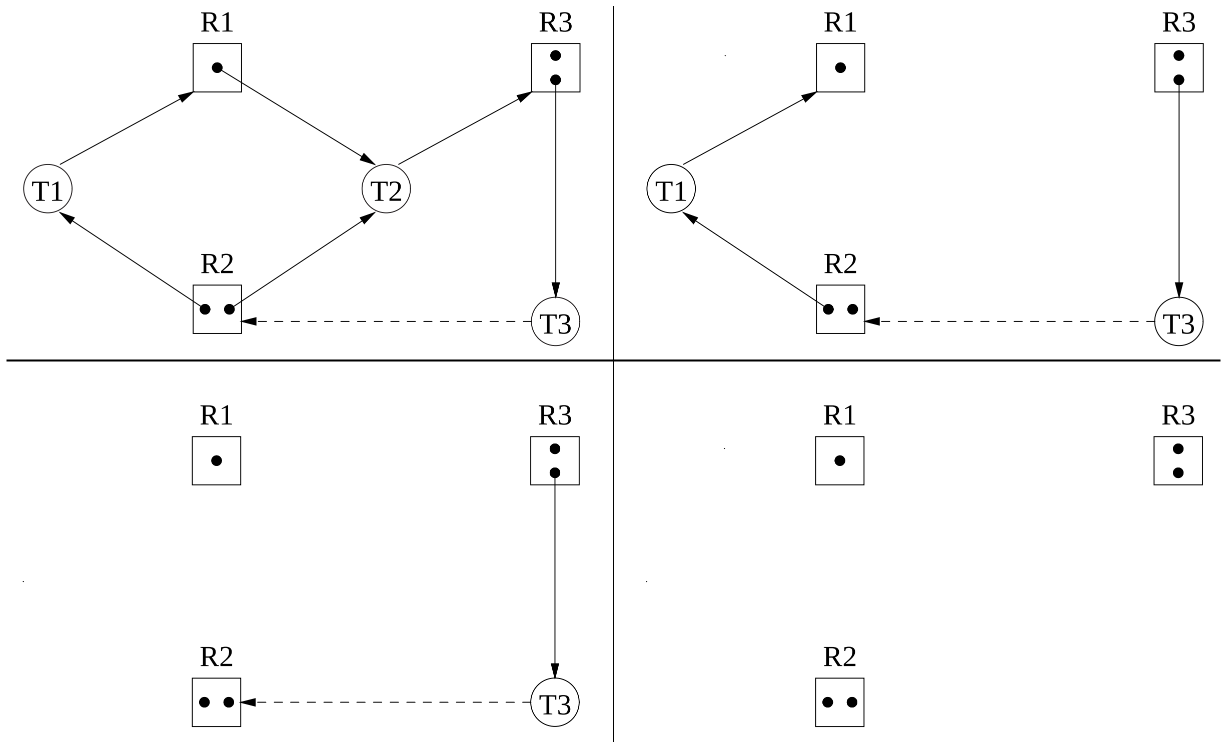
  T1 $\rightarrow$ R1 $\rightarrow$ T2 $\rightarrow$ R3 $\rightarrow$ T3 $\rightarrow$ R2 $\rightarrow$ T1 (cycle)
  T2 $\rightarrow$ R3 $\rightarrow$ T3 $\rightarrow$ R2 $\rightarrow$ T2 (cycle)

  - If T4 releases its resource, the cycle is broken.

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



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

7.5 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.

7.6 Which Method To Chose?

- Maybe "none of the above": just ignore the problem
  - if some process is blocked for rather a long time, assume it is deadlocked and abort it
  - do this automatically in transaction-processing systems, manually elsewhere
- Of the techniques studied, only the ordered resource policy turns out to have much practical value.

8 Indirect Communication

- P and V are low level primitives for protecting critical sections and establishing synchronization between tasks.
- Shared variables provide the actual information that is communicated.
- 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.

8.1 Critical Regions

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

```
VAR v : SHARED INTEGER MutexLock v_lock;
```

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

```
REGION v DO                    v_lock.acquire()
    // critical section            ... // x = v; (read)  v = y (write)
END REGION                     v_lock.release()
```

- Nesting can result in deadlock.

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

task$_1$ / **task**$_2$

```
task1                  task2
REGION x DO            REGION y DO
    . . .                  . . .
    REGION y DO            REGION x DO
        . . .                  . . .
    END REGION             END REGION
    . . .                  . . .
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.

8.2 Conditional Critical Regions

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

```
REGION v DO
    AWAIT conditional-expression
    ...
END REGION
```

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

```
VAR Q : SHARED QUEUE<INT,10>

REGION Q DO
    AWAIT NOT EMPTY( Q ) buffer not empty
    take an item from the front of the queue
END REGION
```

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

8.3 Monitor

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

```
_Monitor name {
    shared data
    members that see and modify the data
};
```

- A **mutex member** (short for mutual-exclusion member) is one that does NOT begin 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.

```
class Mon {
    MutexLock mlock;
    int v;
  public:
    int x(...) {
        mlock.acquire();
        ...          // int temp = v;
        mlock.release();
        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:

```
_Monitor AtomicCounter {
    int counter;
  public:
    AtomicCounter( int init = 0 ) : 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.

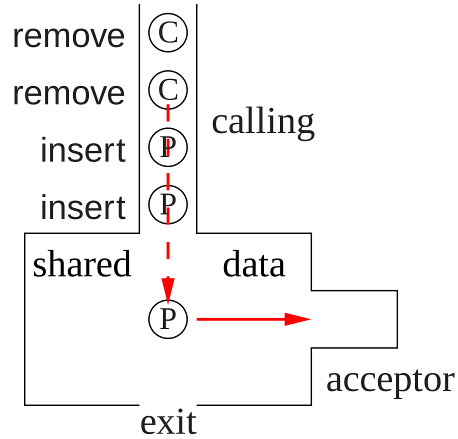
8.4 Scheduling (Synchronization)

- A monitor may want to schedule tasks in an order different from the order in which they arrive (bounded buffer, readers/write with staleness/freshness).
- There are two techniques: external and internal scheduling.
  - external is scheduling tasks outside the monitor and is accomplished with the accept statement.
  - internal is scheduling tasks inside the monitor and is accomplished using condition variables with signal & wait.

8.4.1 External Scheduling

- 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.
- Each **_Accept** defines what cooperation must occur for the accepting task to proceed.
- E.g. Bounded Buffer

```
_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 );
    elements[back] = elem;
    back = ( back + 1 ) % 20;
    count += 1;
}
int BoundedBuffer::remove() {
    if ( count == 0 ) _Accept( insert );
    int elem = elements[front];
    front = ( front + 1 ) % 20;
    count -= 1;
    return elem;
}
```



- 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 waits), the acceptor continues.
- If the accepted task does an accept, it blocks, forming a stack of blocked acceptors.
- External scheduling is simple because unblocking (signalling) is implicit.

8.4.2 Internal Scheduling

- Scheduling among tasks inside the monitor.
- A **condition** is a queue of waiting tasks:

```
uCondition x, y, z[5];
```

- A task waits (blocks) by placing itself on a condition:

```
x.wait();      // 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.

- empty returns **false** if there are tasks blocked on the queue and **true** otherwise.

- front returns an integer value stored with the waiting task at the front of the condition queue.

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

  ```
  x.signal();
  ```

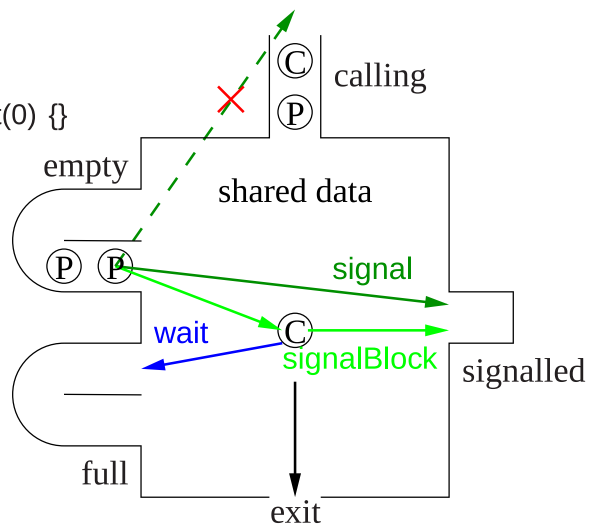
  removes a blocked task at the front of the condition queue and makes it ready.

- Signaller does not block, so the signalled task must continue waiting until the signaller exits or waits.

- Like a SyncLock, a signal on an empty condition is lost!

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

```
_Monitor BoundedBuffer {
    uCondition full, empty;
    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 ) empty.wait();
        elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        full.signal();
    }
    int remove() {
        if ( count == 0 ) full.wait();
        int elem = elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        empty.signal();
        return elem;
    }
};
```



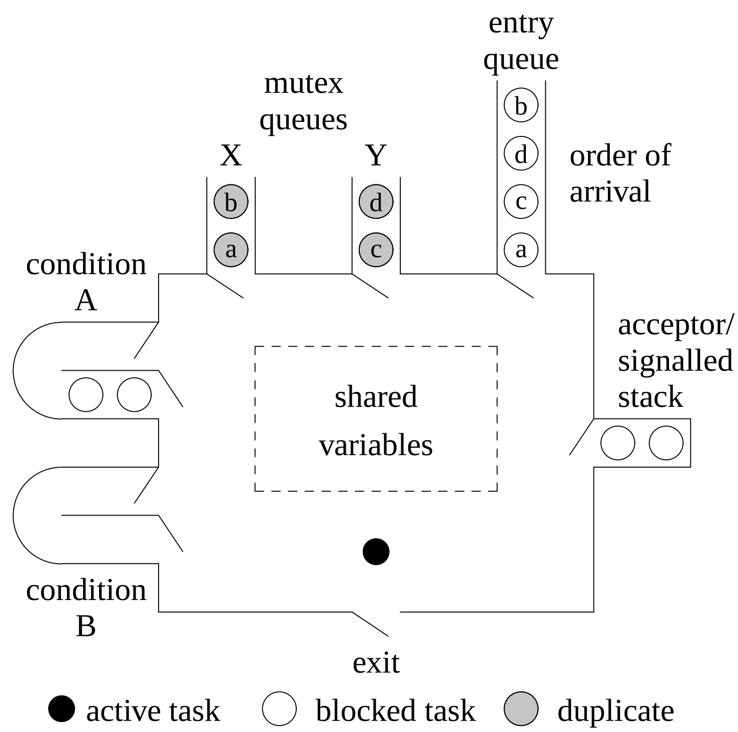
- **wait()** blocks the current thread, and restarts a signalled task or implicitly releases the monitor lock.

- **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

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



- **entry queue** is FIFO list of calling tasks to the monitor.
- When to use external or internal scheduling?
- External is easier to specify and explain over internal with condition variables.
- However, external scheduling cannot be used if:
  - scheduling depends on member parameter value(s), e.g., compatibility code for dating
  - scheduling must block in the monitor but cannot guarantee the next call fulfils cooperation, e.g., if boy accepts girl, she may not match (and boys cannot match).

```
_Monitor DatingService {
    uCondition girls[20], boys[20], exchange;
    int girlPhoneNo, boyPhoneNo;
  public:
    int girl( int phoneNo, int ccode ) {
        if ( boys[ccode].empty() ) {
            girls[ccode].wait();
            girlPhoneNo = phoneNo;
            exchange.signal();
        } else {
            girlPhoneNo = phoneNo;
            boys[ccode].signal(); // signalBlock() & remove exchange
            exchange.wait();
        }
        return boyPhoneNo;
    }
    int boy( int phoneNo, int ccode ) {
        // same as above, with boy/girl interchanged
    }
};
```

8.5 Readers/Writer

- E.g. Readers and Writer Problem (Solution 3, Section 6.4.4.3, p. 112, 5 rules but staleness)

```
_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                  writers
rw.startRead()           rw.startWrite()
// read                  // write
rw.endRead()             rw.endWrite()
```

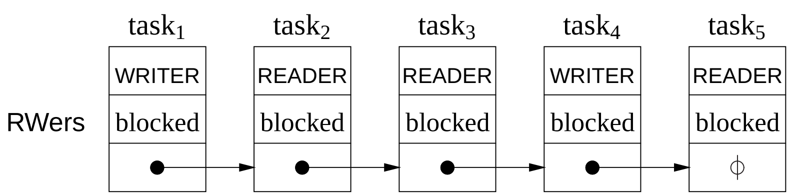
- Alternative interface:

```
_Monitor ReadersWriter {
    _Mutex void startRead();
    _Mutex void endRead();
    _Mutex void startWrite();
    _Mutex void endWrite();
  public:
    _Nomutex void read(...) {
        startRead();
        // read
        endRead();
    }
    _Nomutex void write(...) {
        startWrite();
        // write
        endWrite();
    }
};
```

- E.g. Readers and Writer Problem (Solution 4, Section 6.4.4.4, p. 112, shadow queue)

```
_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 8)

```
    _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?
- **Nested monitor problem**: acquire monitor X, call to monitor Y, and wait on condition in Y.
- Monitor Y's mutex lock is released by wait but monitor X's monitor lock is NOT released ⇒ potential deadlock.

8.6 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

- Possible to simulate P and V using a monitor:

```
_Monitor semaphore {
    int sem;
    uCondition semcond;
  public:
    semaphore( int cnt = 1 ) : sem( cnt ) {}
    void P() {
        if ( sem == 0 ) semcond.wait();
        sem -= 1;
    }
    void V() {
        sem += 1;
        semcond.signal();
    }
};
```

- Can this simulation be reduced?

8.7 Monitor Types

- **explicit scheduling** occurs when:
  - 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.

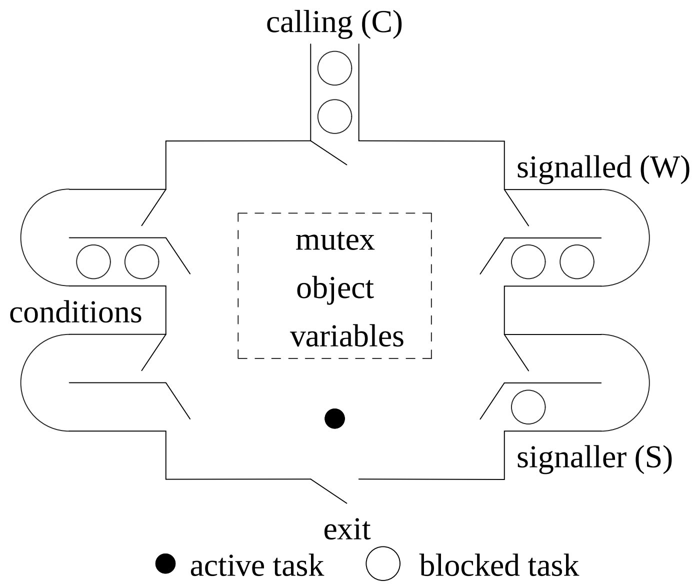
- **implicit scheduling** occurs when a task waits in or exits from a mutex member, and a new task is selected first from the A/S stack, then the entry queue.

-

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

- 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).

	relative priority	
1	$C = W = S$	
2	$C = W < S$	
3	$C = S < W$	
4	$C < W = S$	
5	$C < W < S$	
6	$C < S < W$	
7	$S = W < C$	rejected
8	$W < S = C$	
9	$W < C < S$	
10	$S < W = C$	
11	$S < C < W$	
12	$W < S < C$	
13	$S < W < C$	

- 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.

```
_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;
    }
};
```

- Additional restricted monitor-type requiring the signaller exit immediately from monitor (i.e., signal ⇒ return), called **immediate-return signal**.
- Ten kinds of useful monitor are suggested:

signal type	priority	no priority
Blocking	Priority Blocking (Hoare) $C < S < W$ (μC++ signalBlock)	No Priority Blocking $C = S < W$
Nonblocking	Priority Nonblocking $C < W < S$ (μC++ signal)	No Priority Nonblocking $C = W < S$ (Java/C#)
Quasi -blocking	Priority Quasi $C < W = S$	No Priority Quasi $C = W = S$
Immediate Return	Priority Return $C < W$	No Priority Return $C = W$
Implicit Signal	Priority Implicit Signal $C < W$	No Priority Implicit Signal $C = W$

  - ○ 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).
- No-priority non-blocking 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**
- No-priority blocking monitors require the signaller task to recheck the waiting condition in case of a barging task.

  ⇒ use a **while** loop around a signal to check for barging

- coroutine monitor (**_Cormonitor**)
  - coroutine with implicit mutual exclusion on calls to specified member routines:

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

  - can use resume(), suspend(), condition variables (wait(), signal(), signalBlock()) or **_Accept** on mutex members.
  - coroutine can now be used by multiple threads, e.g., coroutine print-formatter accessed by multiple threads.

8.8 Java Monitor

- Java has **synchronized** class members (i.e., _Mutex members but incorrectly named), and a **synchronized** statement.
- All classes have one implicit condition variable and these routines to manipulate it:

```
public wait();
public notify();
public notifyAll()
```

- Java concurrency library has multiple conditions but incompatible with language condition (see Section 11.5.1, p. 197).
- Internal scheduling is no-priority nonblocking ⇒ barging
  - wait statements must be in while loops to recheck conditions.

- Bounded buffer:

```
class buffer {
    // buffer declarations
    private int count = 0;
    public synchronized void insert( int elem ) {
        while ( count == Size ) wait(); // busy-waiting
        // add to buffer
        count += 1;
        if ( count == 1 ) notifyAll();
    }
    public synchronized int remove() {
        while ( count == 0 ) wait();    // busy-waiting
        // remove from buffer
        count -= 1;
        if ( count == Size - 1 ) notifyAll();
        return elem;
    }
}
```

- Only one condition queue, producers/consumers wait together ⇒ unblock all tasks.

- Only one condition queue ⇒ certain solutions are difficult or impossible.

- Erroneous Java implementation of barrier:

```
class Barrier {                         // monitor
    private int N, count = 0;
    public Barrier( int N ) { this.N = N; }
    public synchronized void block() {
        count += 1;                     // count each arriving task
        if ( count < N )
            try { wait(); } catch( InterruptedException e ) {}
        else                            // barrier full
            notifyAll();                // wake all barrier tasks
        count -= 1;                     // uncount each leaving task
    }
}
```

  - Nth task does notifyAll, leaves monitor and performs its ith step, and then races back (barging) into the barrier before any notified task restarts.
  - It sees count still at N and incorrectly starts its ith+1 step before the current tasks have completed their ith step.

- Fix by modifying code for Nth task to set count to 0.

```
    else {                      // barrier full
        count = 0;              // reset count
        notifyAll();            // wake all barrier tasks
    }
```

- Technically, still wrong because of **spurious wakeup** ⇒ require loop around wait.

```
if ( count < N )
    while ( ??? )  // cannot be count < N as count is always < N
        try { wait(); } catch( InterruptedException e ) {}
```

- Requires more complex implementation.

```
class Barrier {                              // monitor
    private int N, count = 0, generation = 0;
    public Barrier( int N ) { this.N = N; }
    public synchronized void block() {
        int mygen = generation;
        count += 1;                          // count each arriving task
        if ( count < N )                     // barrier not full ? => wait
            while ( mygen == generation )
                try { wait(); } catch( InterruptedException e ) {}
        else {                               // barrier full
            count = 0;                       // reset count
            generation += 1;                 // next group
            notifyAll();                     // wake all barrier tasks
        }
    }
}
```

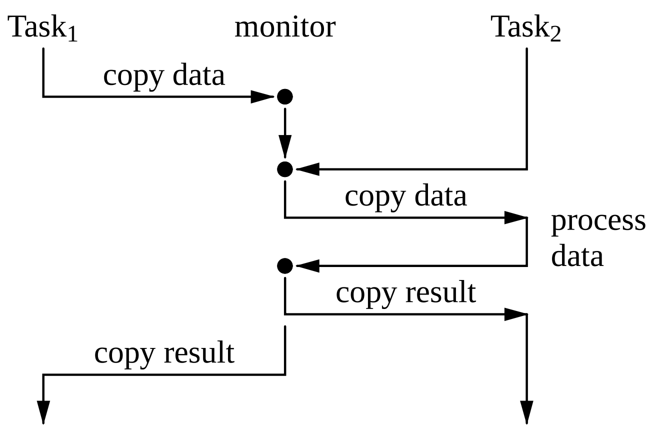
- Misconception of building condition variables in Java with nested monitors:

```
class Condition {                            // try to build condition variable
    public synchronized void Wait() {
         try { wait(); } catch( InterruptedException ex ) {};
    }
    public synchronized void Notify() { notify(); }
}
class BoundedBuffer {
    // buffer declarations
    private Condition full = new Condition(), empty = new Condition();
    public synchronized void insert( int elem ) {
        while ( count == NoOfElems ) empty.Wait(); // block producer
        // add to buffer
        count += 1;
        full.Notify();                       // unblock consumer
    }
    public synchronized int remove() {
        while ( count == 0 ) full.Wait(); // block consumer
        // remove from buffer
        count -= 1;
        empty.Notify();                      // unblock producer
        return elem;
    }
}
```

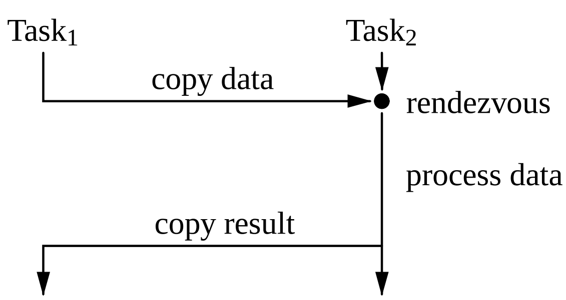
- Deadlocks at empty.Wait()/full.Wait() as buffer monitor-lock is not released.

9 Direct Communication

- Monitors work well for passive objects that require mutual exclusion because of sharing.
- However, communication among tasks with a monitor is indirect.
- Problem: point-to-point with reply indirect communication:



- Point-to-point with reply direct communication:



- Tasks can communicate directly by calling each others member routines.

9.1 Task

- A task is 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.
- A task is like a monitor because it provides mutual exclusion (and synchronization) so only one thread is active in the object.
  - public members of a task are implicitly mutex and other kinds of members can be made explicitly mutex.

- ○ external scheduling allow direct calls to mutex members (task's thread blocks while caller's executes)
- ○ without external scheduling, tasks must *call out* to communicate $\Rightarrow$ third party, or somehow emulate external scheduling with internal.

- In general, basic execution properties produce different abstractions:

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

- When thread or stack is missing it comes from calling object.
- Abstractions are not ad-hoc, rather derived from basic properties.
- Each of these abstractions has a particular set of problems it can solve, and therefore, each has a place in a programming language.

9.2 Scheduling

- A task may want to schedule access to itself by other tasks in an order different from the order in which requests arrive.
- As for monitors, there are two techniques: external and internal scheduling.

9.2.1 External Scheduling

- As for a monitor (see Section 8.4.1, p. 131), the accept statement can be used to control which mutex members of a task can accept calls.

```
_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 ( ;; ) {        // INFINITE LOOP!!!
                _When (count != 20) _Accept(insert) {  // after call
                } or _When (count != 0) _Accept(remove) { // after call
                } // _Accept
            } // for
        }
};
```

- Why are accept statements moved from member routines to the task main?
- **_Accept**(m1, m2) S1 $\equiv$ **_Accept**(m1) S1; **or** **_Accept**(m2) S1;

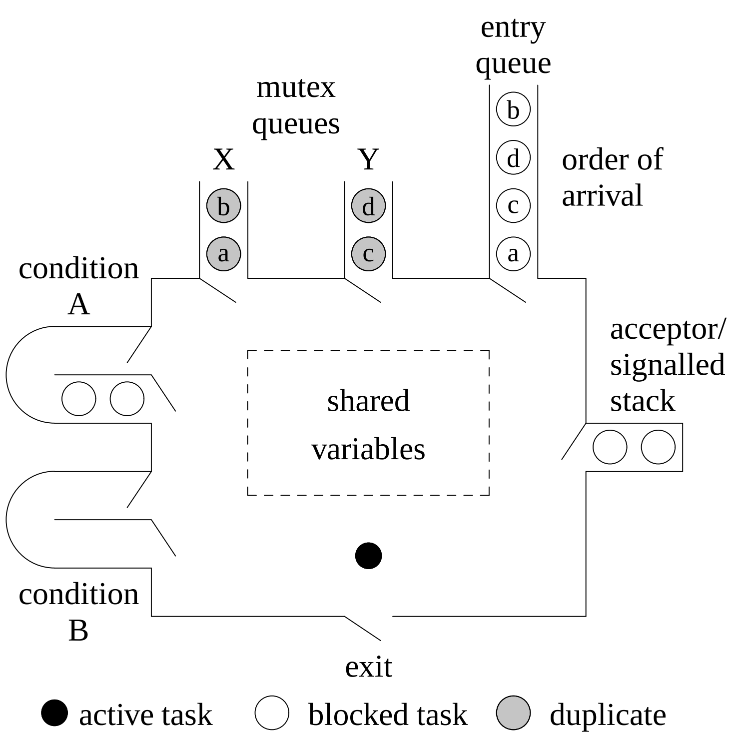
  if (C1 || C2) S1 $\equiv$ **if** (C1) S1; **else if** (C2) S1;
- Extended version allows different **_When** and code after call for each accept.
- Equivalence using **if** statements:

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

- Generalize from 2 to 3 conditionals/members:

```
if      ( C1 && C2 && C3 ) _Accept( M1, M2, M3 );
else if ( C1 && C2 ) _Accept( M1, M2 );
else if ( C1 && C3 ) _Accept( M1, M3 );
else if ( C2 && C3 ) _Accept( M2, M3 );
else if ( C1 ) _Accept( M1 );
else if ( C2 ) _Accept( M2 );
else if ( C3 ) _Accept( M3 );
```

- Necessary to ensure that for every true conditional, only the corresponding members are accepted.
- $2^N - 1$ **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 **_Accept**s.
  - Hence, the order of the **_Accept**s indicates their relative priority for selection if there are several outstanding calls.

- Once the accepted call has completed *or the caller waits*, the statement after the accepting **_Accept** clause is executed and the accept statement is complete.

  ◦ To achieve greater concurrency in the bounded buffer, change to:

```
    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 ( uMutexFailure::RendezvousFailure ) {
            // deal with rendezvous failure
        } // try
    }
};
```

9.2.2 Accepting the Destructor

- Common way to terminate a task is to have a stop member:

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

- Call stop when task is to stop:

```
void uMain::main() {
    BoundedBuffer buf;
    // create producer & consumer tasks
    // delete producer & consumer tasks
    buf.stop();  // 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:

```
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.

9.2.3 Internal Scheduling

- Scheduling among tasks inside the monitor.
- As for monitors, condition, signal and wait are used.

```
_Task BoundedBuffer {
    uCondition full, empty;
    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 ) empty.wait();
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        full.signal();
    }
```

```
    int remove() {
        if ( count == 0 ) full.wait();
        int elem = Elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        empty.signal();
        return elem;
    }
  protected:
    void main() {
        for ( ;; ) {
            _Accept( ~BoundedBuffer )
                break;
            or _Accept( insert, remove );
            // do other work
        } // for
    }
};
```

- Is there a potential starvation problem?

9.3 Increasing Concurrency

- 2 task involved in direct communication: client (caller) & server (callee)
- possible to increase concurrency on both the client and server side

9.3.1 Server Side

- Server manages a resource and server thread should introduce additional concurrency (assuming no return value).

No Concurrency	Some Concurrency

`_Task server1 {`
`  public:`
`    void mem1(...) { S1 }`
`    void mem2(...) { S2 }`
`    void main() {`
`        ...`
`        _Accept( mem1 );`
`        or _Accept( mem2 );`
`    }`
`}`
	`_Task server2 {`
	`  public:`
	`    void mem1(...) { S1.copy-in }`
	`    void mem2(...) { S2.copy-out }`
	`    void main() {`
	`        ...`
	`        _Accept( mem1 ) { S1.work }`
	`        or _Accept( mem2 ) { S2.work };`
	`    }`
	`}`

- No concurrency in left example as server is blocked, while client does work.
- Alternatively, client blocks in member, server does work, and server unblocks client.
- No concurrency in either case, only mutual exclusion.
- Some concurreny possible in right example if service can be factored into administrative (S1.copy) and work (S1.work) code.

  - i.e., move code from the member to statement executed after member is accepted.

- Small overlap between client and server (client gets away earlier) increasing concurrency.

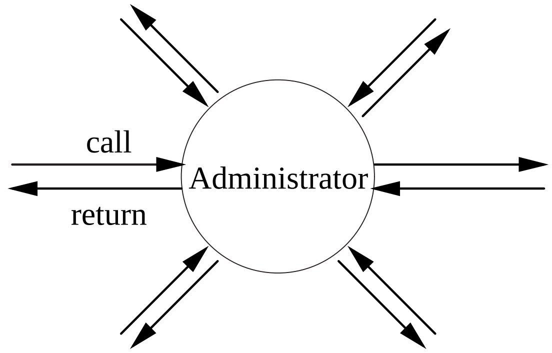
9.3.1.1 Internal Buffer

- The previous technique provides buffering of size 1 between the client and server.
- Use a larger internal buffer to allow clients to get in and out of the server faster?
- 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.
- Only way to free server's thread to receive new requests and return finished results to clients is add another thread.
- Additional thread is a **worker task** that calls server to get work from buffer and return results to buffer.
- Note, customer (client), manager (server) and employee (worker) relationship.
- Number of workers has to balance with number of clients to maximize concurrency (bounded-buffer problem).

9.3.1.2 Administrator

- An **administrator** is a server managing multiple clients and worker tasks.
- The key is that an administrator does little or no "real" work; its job is to manage.
- 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 calling may block the administrator.
- An administrator usually maintains a list of work to pass to **worker tasks**.
- Typical workers are:

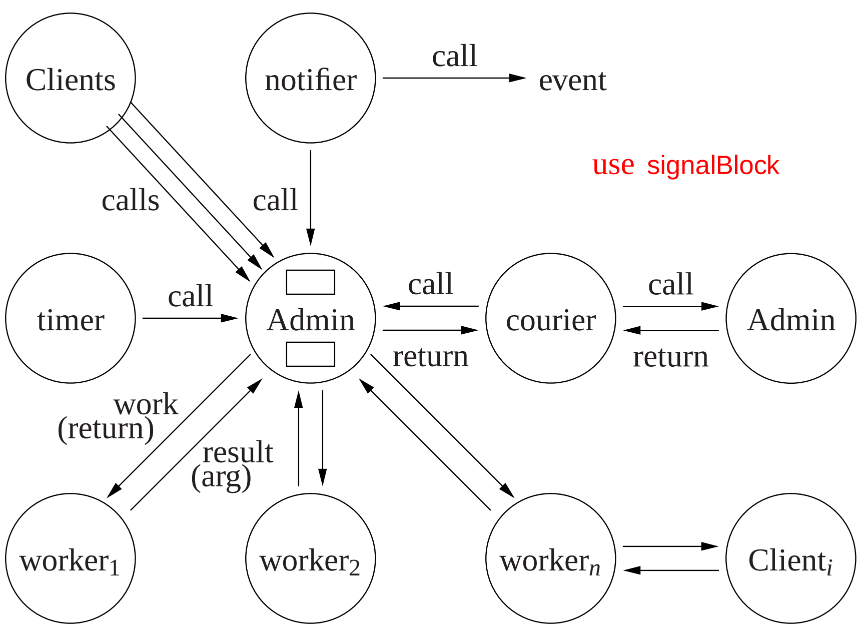
timer - prompt the administrator at specified time intervals

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



9.3.2 Client Side

- While a server can attempt to make a client's delay as short as possible, not all servers do it.

- In some cases, a client may not have to wait for the server to process a request (producer/consumer problem)
- This can be accomplished by an asynchronous call from the client to the server, where the caller does not wait for the call to complete.
- 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.

9.3.2.1 Returning Values

- If a client only drops off data to be processed by the server, the asynchronous call is simple.
- However, if a result is returned from the call, i.e., from the server to the client, the asynchronous call is significantly more complex.
- To achieve asynchrony in this case, a call must be divided into two calls:

```
callee.start( arg );            // provide arguments
// caller performs other work asynchronously
result = callee.finish();       // obtain result
```

- 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.

9.3.2.2 Tickets

- One form of protocol is the use of a token or ticket.
- The first part of the protocol transmits the arguments specifying the desired work and a ticket (like a laundry ticket) is returned immediately.
- The second call 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).

9.3.2.3 Call-Back Routine

- Another protocol is to transmit (register) a routine on the initial call.
- When the result is ready, the routine is called by the task generating the result, passing it the result.
- The call-back routine cannot block the server; it can only store the result and set an indicator (e.g., V a semaphore) known to the 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.

9.3.2.4 Futures

- A **future** provides the same asynchrony as above but without an explicit protocol.
- The protocol becomes implicit between the future and the task generating the result.
- 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.

```
future = callee.work( arg );        // provide arguments, return future
// perform other work asynchronously
i = future + . . .;                 // obtain result, may block if not ready
```

- 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.
- The general design for a future is:

```
class Future : public ResultType {
    friend _Task server;    // allow server to access internal state
    ResultType result;
    uSemaphore avail;
    Future *link;
  public:
    Future() : avail( 0 ) {}

    ResultType get() {
        avail.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.
- μC++ provides two forms of template futures, which differ in storage management.
  - Explicit-Storage-Management future (Future_ESM<T>) must be allocated and deallocated explicitly by the client.
  - Implicit-Storage-Management future (Future_ISM<T>) automatically allocates and frees storage (when future no longer in use, GC).
- Focus on Future_ISM as simpler to use but less efficient is certain cases.
- Basic set of operations for both types of futures, divided into client and server operations.

Client

- Future value:

```
#include <uFuture.h>
Server server;                    // server thread handles async calls
Future_ISM<int> f[10];
for ( int i = 0; i < 10; i += 1 ) {
    f[i] = server.perform( i );   // async call to server
}
// work asynchronously while server processes requests
for ( int i = 0; i < 10; i += 1 ) {  // retrieve async results
    osacquire( cout ) << f[i]() << " " // synchronize on retrieve value
        << (int)f[i] << " " << f[i] << endl; // use value again (cheap access)
}
f[3] = 3;      // DISALLOWED: OTHER THREADS CAN READ VALUE
```

• Future pointer:

```
#include <uFuture.h>
Server server;                  // server thread handles async calls
int val
Future_ISM<int *> fval;
fval = server.perform( val );   // async call to server (change val by reference)
// work asynchronously while server processes requests
osacquire( cout ) << *fval() << endl;  // synchronize on retrieve value
val = 3;      // ALLOWED: BUT FUTURE POINTER IS STILL READ-ONLY
```

available – returns **true** if the asynchronous call has completed and **false** otherwise.

complete ⇒ result available, server raised an exception, or call is cancelled

operator() – (function call) returns a ***read-only*** copy of the future result.

block if the future result unavailable; raise exception if exception returned by server.

future result can be retrieved multiple times by any task (⇒ read-only) until the future is reset or destroyed.

operator T – (conversion to type T) returns a ***read-only*** copy of the future result.

***Can only be performed only after a blocking access or a call to* available *returns* true.**

A low-cost way to get future result *after* the result is delivered; raise exception if exception returned by server.

cancelled – returns **true** if the future is cancelled and **false** otherwise.

cancel – attempts to cancel the asynchronous call the future refers to.

Clients waiting for the result are unblocked, and exception of type Future_ESM::Cancellation is raised at them.

Server

```
struct Work {
    int i;                          // argument(s)
    Future_ISM<int> result;         // result
    Work( int i ) : i( i ) {}
};
Future_ISM<int> server::perform( int i ) {  // called by clients
    Work *w = new Work( i );        // create work request
    requests.push_back( w );        // add to list of requests
    return w->result;               // return future in request
}
// server or server's worker does
Work *w = requests.front();         // take next work request
requests.pop_front();               // remove request
int r = . . .                       // compute result using argument w.i
result.reset();                     // possibly reset future
w->result.delivery( r );            // insert result into future
```

delivery(T result) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

reset – mark the future as empty so it can be reused, after which the current future value is no longer available.

exception(uBaseEvent *cause) – copy a server-generated exception into the future, and the exception cause is thrown at waiting clients.

```
_Event E {};
Future_ISM<int> result;
result.exception( new E ); // deleted by future
```

exception deleted by reset or when future deleted

Complex Future Access

- **select statement** waits for one or more heterogeneous futures based on logical selection-criteria.
- Simplest select statement has a single **_Select** clause, e.g.:

```
_Select( selector-expression );
```

- Selector-expression must be satisfied before execution continues.
- For a single future, the expression is satisfied if and only if the future is available.

```
_Select( f1 );    ≡    f1();
```

- Selector is select blocked until f1.available() is true (equivalent to calling future access-operator).
- Multiple futures may appear in a compound selector-expression, related using logical operators || and &&:

```
_Select( f1 || f2 && f3 );
```

- Normal operator precedence applies: **_Select**(**(** f1 || **(** f2 && f3 **))**).
- Execution waits until either future f1 is available or both futures f2 and f3 are available.
- For any selector-expression containing an || operator, some futures in the expression may be unavailable after the selector-expression is satisfied.
- E.g., in the above, if future f1 becomes available, neither, one or both of f2 and f3 may be available.

- A **_Select** clause may be guarded with a logical expression and have code executed after a future receives a value:

```
_When ( conditional-expression ) _Select( f1 )
    statement-1                    // action
or
    _When ( conditional-expression ) _Select( f2 )
        statement-2                // action
    and _When ( conditional-expression ) _Select( f3 )
        statement-3                // action
```

- **or** and **and** keywords relate the **_Select** clauses like operators || and && relate futures in a select-expression, including precedence.

```
_Select( f1 )          ≡     _Select( f1 || f2 && f3 );
or _Select( f2 )
and _Select( f3 );
```

- Parentheses may be used to specify evaluation order.

```
( _Select( f1 )          ≡     _Select( ( f1 || ( f2 && f3 ) )
or ( _Select( f2 )
and _Select( f3 ) ) );
```

- Each **_Select**-clause action is executed when its sub-selector-expression is satisfied, i.e., when each future becomes available.

- However, control does not continue until the selector-expression associated with the entire statement is satisfied.

- E.g., if f2 becomes available, statement-2 is executed but the selector-expression associated with the entire statement is not satisfied so control blocks again.

- When either f1 or f3 become available, statement-1 or 3 is executed, and the selector-expression associated with the entire statement is satisfied so control continues.

- Within the action statement, it is possible to access the future using the non-blocking access-operator since the future is known to be available.

- An action statement is triggered only once for its selector-expression, even if the selector-expression is compound.

```
_Select( f1 || f2 )
    statement-1  // triggered once
and _Select( f3 )
    statement-2
```

- In statement-1, it is unknown which of futures f1 or f2 satisfied the sub-selector-expression and caused the action to be triggered.

- Hence, it is necessary to check which of the two futures is available.

- A select statement can be non-blocking using a terminating **_Else** clause, e.g.:

```
_Select( selector-expression )
    statement                       // action
_When ( conditional-expression ) _Else  // terminating clause
    statement                       // action
```

- The **_Else** clause *must* be the last clause of a select statement.
- If its guard is true or omitted and the select statement is not immediately true, then the action for the **_Else** clause is executed and control continues.
- If the guard is false, the select statement blocks as if the **_Else** clause is not present.

10 Optimization

- A computer with infinite memory and speed requires no optimizations to use less memory or run faster (space/time).
- With finite resources, optimization is useful/necessary to conserve resources and for good performance.
- General forms of optimizations are:
  - **reordering**: data and code are reordered to increase performance in certain contexts.
  - **eliding**: removal of unnecessary data, data accesses, and computation.
  - **replication**: processors, memory, data, code are duplicated because of limitations in processing and communication speed (speed of light).
- Optimized program must be isomorphic to original $\Rightarrow$ produce same result for fixed input.
- Kinds of optimizations are restricted by the kind of execution environment.

10.1 Sequential Execution

- ***Sequential*** execution presents simple semantics for optimization.
  - operations occur in **program order**, i.e., sequentially
- Dependencies result in partial ordering among a set of statements:
  - **data dependency** (R $\Rightarrow$ read, W $\Rightarrow$ write)

$R_x \rightarrow R_x$	$W_x \rightarrow R_x$	$R_x \rightarrow W_x$	$W_x \rightarrow W_x$
`y = x;` `z = x;`	`x = 0;` `y = x;`	`y = x;` `x = 3;`	`x = 0;` `x = 3;`

    Which statements cannot be reordered?
  - **control dependency**

    ```
    1   if ( x == 0 )
    2       y = 1;
    ```

    Statements cannot be reordered as line 1 determines if 2 is executed.
- Most programs are not written in optimal order or in minimal form.
  - OO, Functional, SE are seldom optimal approaches on von Neumann machine.
- To achieve better performance, compiler/hardware make changes:
  1. reorder disjoint (independent) operations (**variables have different addresses**)

$R_x \rightarrow R_y$	$W_x \rightarrow R_y$	$R_x \rightarrow W_y$	$W_x \rightarrow W_y$
`t = x;` `s = y;`	`x = 0;` `y == 1;`	`x == 1;` `y = 3;`	`y = 0;` `x = 3;`

Which statements cannot be reordered?

2. elide unnecessary operations (transformation/dead code)

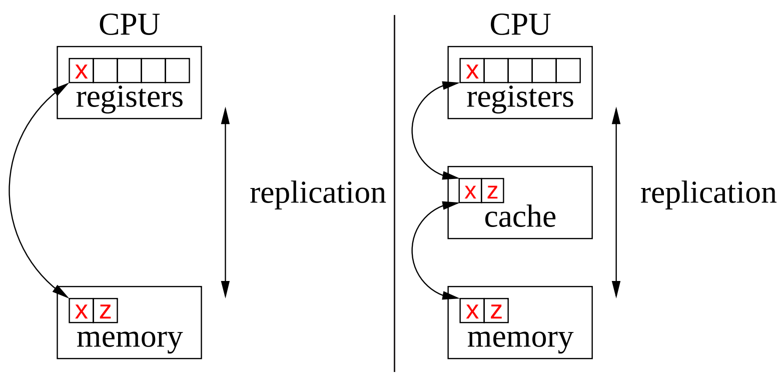
```
x = 0;  // unnecessary, immediate change
x = 3;

for ( int i = 0; i < 10000; i += 1 ); // unnecessary, no loop body

int factorial( int n, int acc ) {  // tail recursion
    if (n == 0) return acc;
    return factorial( n - 1, n * acc ); // convert to loop
}
```

3. execute in parallel if multiple functional-units (adders, floating units, pipelines, cache)

- Very complex reordering, reducing, and overlapping of operations allowed.
- Overlapping implies parallelism, ***but limited capability in sequential execution.***

10.2 Memory Hierarchy

- Complex memory hierarchy:



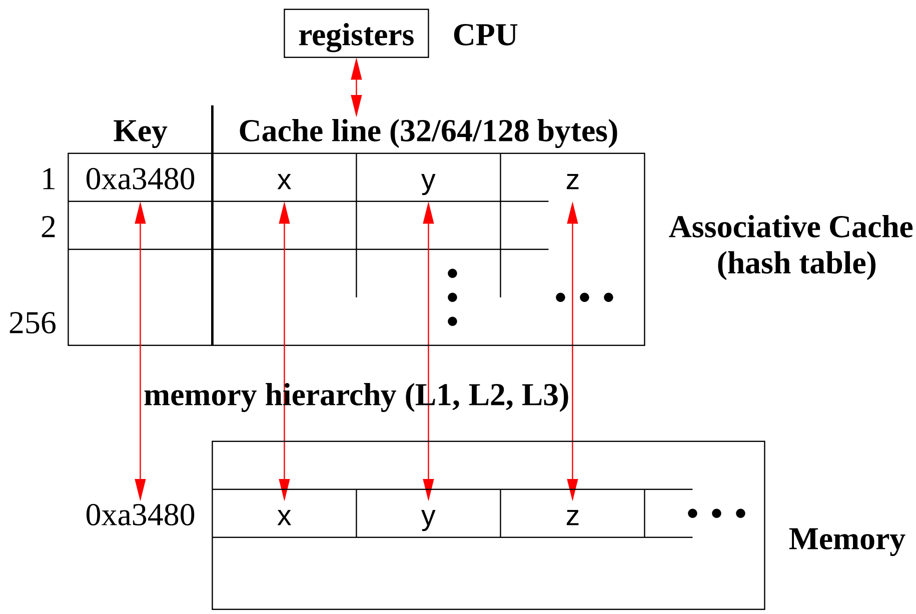
- Optimizing data flow along this hierarchy defines a computer's speed.
- Hardware aggressively optimizes data flow for sequential execution.
- Having basic understanding of cache is essential to understanding performance of both sequential and concurrent programs.

10.2.1 Cache Review

- Problem: CPU 100(0) times faster than memory (100,00(0) times faster than disk).
- Solution: copy data from general memory into very, very fast local-memory (registers).
- Problem: billions of bytes of memory but only 6–256 registers.
- Solution: move highly accessed data ***within*** a program from memory to registers for as long as possible and then back to memory.

- Problem: quickly run out of registers as more data accessed.
  - ⇒ must rotate data from memory through registers dynamically.
  - compiler attempts to keep highly used variables in registers (LRU, requires Oracle)
- Problem: does not handle highly accessed data ***among*** programs (threads).
  - each context switch saves and restores most registers to memory
- Solution: use hardware **cache** (automatic registers) to stage data without pushing to memory and allow sharing of data among programs.

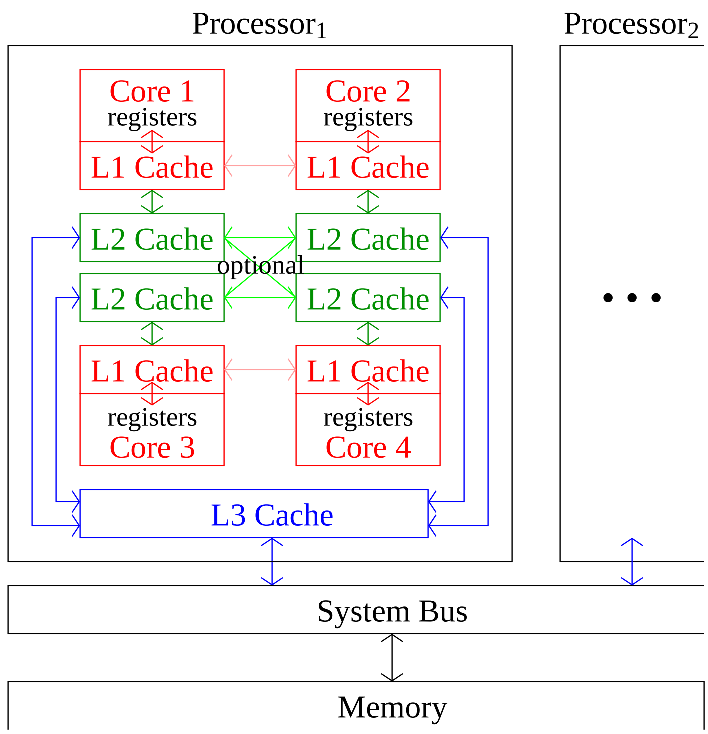
```
int x, y, z;
x += 1;     ld   r1,0xa3480     // load register 1 from x
            add r1,#1           // increment
            st   r1,0xa3480     // store register 1 to x
```



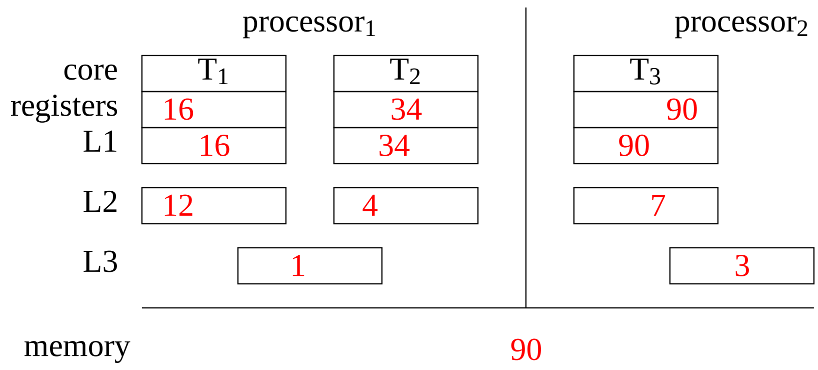
  - Caching transparently hides the latency of accessing main memory.
  - Cache loads in 32/64/128 bytes, called **cache line**, with addresses multiple of line size.
  - When x is loaded into register 1, a cache line containing x, y, and z are implicitly copied up the memory hierarchy from memory through caches.
  - When cache is full, data evicted, i.e., remove old cache-lines to bring in new (LRU).
  - When program ends, its addresses are flushed from the memory hierarchy.
- In theory, cache can eliminate registers, but registers provide small addressable area (register window) with short addresses (3-8 bits for 8-256 registers) ⇒ shorter instructions.

10.2.2 Cache Coherence

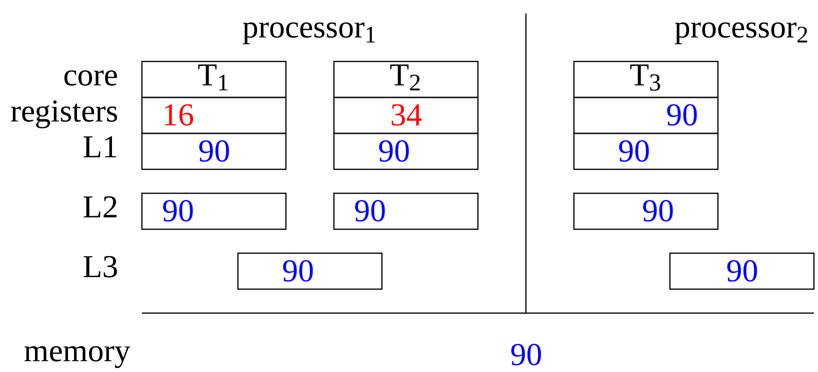
- Multi-level caches used, each larger but with diminishing speed (and cost).
- E.g., 64K L1 cache (32K Instruction, 32K Data) per core, 256K L2 cache per core, and 8MB L3 cache shared across cores.



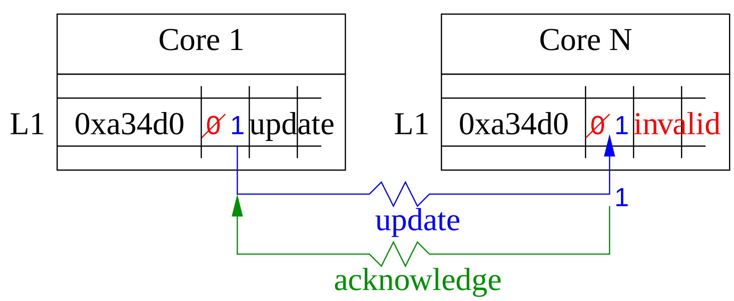
- Data reads logically percolate variables from memory up the memory hierarchy, making cache copies, to registers.
- Why is it necessary to eagerly move reads up the memory hierarchy?
- Data writes from registers to variables logically percolate down the memory hierarchy through cache copies to memory.
- Why is it advantageous to lazily move writes down the memory hierarchy?
- If OS moves program to another processor, all caching information is invalid and the program's data-hierarchy reforms.
- Unlike registers, *all* cache values are shared across the computer.
- Hence, variable can be replicated in a large number of locations.
- No cache coherence for shared variable x (madness)



- With cache coherence for shared variable x



- **Cache coherence** is hardware protocol ensuring update of duplicate data.
- **Cache consistency** addresses ***when*** processor sees update ⇒ bidirectional synchronization.



- Eager cache-consistency means data changes appear instantaneous by waiting for acknowledge from all cores (complex/expensive).
- Lazy cache-consistency allows reader to see own write before acknowledgement ⇒ concurrent programs read stale data!
- If threads continually read/write same memory locations, they invalidate duplicate cache lines, resulting in excessive cache updates, called **cache thrashing**.
  - updated value bounces from one cache to the next

- Because cache line contains multiple variables, cache thrashing can occur inadvertently, called **false sharing**.
- Thread 1 read/writes x while Thread 2 read/writes y ⇒ no direct shared access, but indirect sharing as x and y share cache line.
  - Fix by separating x and y with sufficient storage (padding) to be in next cache line.
  - Difficult for dynamically allocated variables as memory allocator positions storage.

thread 1	thread 2
int *x = **new int**	**int** *y = **new int**;

    x and y may or may not be on same cache line.

10.3 Concurrent Execution

- Concurrent execution
  - non-preemptive (co-operative) scheduling: deterministic in principle, only at routine calls
  - time-slicing: non-deterministic, may increase execution time, but necessary for correctness (stopping busy loop)
- Concurrent program is
  - (large) sections of sequential code per thread connected by
  - small sections of concurrent code where threads interact (protected by synchronization and mutual exclusion (SME))
- In sequential execution, strong memory ordering: reading always returns last value written.
- In concurrent execution, weak memory ordering: reading can return previously written value or value written in future.
  - Happens on multi-processor because of scheduling and buffering (see scrambling/flickering in Section 5.19.6, p. 75 and freshness/staleness in Section 6.4.4.4, p. 112).
  - Notion of ***current*** value becomes blurred for shared variables unless everyone can see values assigned simultaneously.
- SME control order and speed of execution, otherwise non-determinism causes random results or failure (e.g., race condition, Section 7.1, p. 121).
- Sequential sections accessing private variables can be optimized normally ***but not across concurrent boundaries***.
- Concurrent sections accessing shared variables can be corrupted by sequential optimizations ⇒ must restrict optimizations to ensure correctness.
- For correctness and performance, identify concurrent code and only restrict *its* optimization.

- What/how to restrict depends on what sequential assumptions are implicitly applied by hardware and compiler (programming language).
- Following examples show how sequential optimizations cause failures in concurrent code.

10.3.1 Disjoint Reordering

- $R_x \rightarrow R_y$ allows $R_y \rightarrow R_x$

  Reordering disjoint reads does not cause problems. Why?

- $W_x \rightarrow R_y$ allows $R_y \rightarrow W_x$
  - In Dekker entry protocol (see Section 5.19.6, p. 75)

```
1  me = WantIn; // W
2  while ( you == WantIn ) { // R
3      . . .
```

```
   temp = you; // R
1  me = WantIn; // W
2  while ( temp == WantIn ) {
3      . . .
```

  both threads read DontWantIn, both set WantIn, both see DontWantIn, and proceed.

- $R_x \rightarrow W_y$ allows $W_y \rightarrow R_x$
  - In synchronization flags (see Section 5.12, p. 67), allows interchanging lines 1 & 3 for Cons:

```
        Cons
1  while ( ! Insert ); // R
2  Insert = false;
3  data = Data; // W
```

```
        Cons
3  data = Data; // W
1  while ( ! Insert ); // R
2  Insert = false;
```

  allows reading of uninserted data

- $W_x \rightarrow W_y$ allows $W_y \rightarrow W_x$
  - In synchronization flags (see Section 5.12, p. 67), allows interchanging lines 1 & 2 in Prod and lines 3 & 4 in Cons:

```
        Prod
1  Data = i; // W
2  Insert = true; // W
```

```
        Prod
2  Insert = true; // W
1  Data = i; // W
```

  allows reading of uninserted data

  - In Peterson's entry protocol, allows interchanging lines 1 & 2:

```
1  me = WantIn; // W
2  ::Last = &me; // W
```

```
2  ::Last = &me; // W
1  me = WantIn; // W
```

  allows race before either task sets its intent and both proceed

- Compiler uses all of these reorderings to break mutual exclusion:

```
lock.acquire()          // critical section      lock.acquire()
// critical section     lock.acquire()           lock.release();
lock.release();         lock.release();          // critical section
```

- moves lock entry/exit after/before critical section because entry/exit variables not used in critical section.

10.3.2 Eliding

- For high-level language, compiler decides when/which variables are loaded into registers.
- Elide reads (loads) by copying (replicating) value into a register:

```
      T1                          T2
...                         register = flag; // one read, auxiliary variable
flag = false // write       while ( register ); // cannot see change by T1
```

- Hence, variable logically disappears for duration in register.
- ⇒ task spins forever in busy loop if R before W.
- Also, elide meaningless sequential code:

```
sleep( 1 ); // unnecessary in sequential program
```

⇒ task misses signal by not delaying

10.3.3 Replication

- Modern processors increase performance by executing multiple instructions in parallel (data flow, precedence graph (see 6.4.1)).
  - internal pool of instructions taken from program order
  - begin simultaneous execution of instructions with inputs
  - collect results from finished instructions
  - feed results back into instruction pool as inputs
  - ⇒ instructions with independent inputs execute out-of-order
  - From sequential perspective, order of stores ***unimportant***, so hardware starts both instruction (can finish in either order).
  - From concurrent perspective, order of stores ***important***; wrong for variable to become visible before being initialized.
- E.g., **double-check locking** for singleton-pattern:

```
int *ip;
...
if ( ip == NULL ) {            // no storage ?
    lock.acquire();            // attempt to get storage (race)
    if ( ip == NULL ) {        // still no storage ? (double check)
        ip = new int( 0 );     // obtain storage
    }
    lock.release();
}
```

Why do the first check? Why do the second check?

- Fails if last two writes become visible in reverse order, i.e., see ip but uninitialized (see **??**).

```
call    malloc    // new storage address returned in r1
st      #0,(r1)   // initialize storage
st      r1,ip     // initialize pointer
```

- How can cache cause double-check locking to fail?

10.4 Memory Model

- CPU manufactures define set of optimizations performed implicitly by processor (CPU).
- Set of optimizations indirectly define a **memory model**.

Relaxation / Model	W → R	R → W	W → W	Lazy cache update
atomic consistent (AT)				
sequential consistency (SC)				√
total store order (TSO)	√			√
partial store order (PSO)	√	√		√
weak order (WO)	√	√	√	√
release consistency (RC)	√	√	√	√

- AT requires all events occur instantaneously ⇒ slow or impossible (distributed systems).
- SC accepts all events cannot occur instantaneously ⇒ may read old values
- SC still strong enough for software mutual-exclusion (Dekker 5.19.6 / Peterson 5.19.7).
  - SC often considered minimum model for concurrency (Java provides SC)
- No hardware supports AT/SC; TSO (x86/SPARC), PSO (ARM), WO (Alpha), RC (PowerPC)

10.5 Preventing Optimization Problems

- All optimization problems result from races on shared variables.
- If shared data is protected by locks (implicit or explicit),
  - locks define the sequential/concurrent boundaries,
  - boundaries can provide preclude optimizations that affect concurrency.
- Called **race free** because programmer's variables do not have races because synchronization and mutual exclusion.
- However, race free does mean there are no races.
- Races are internal to locks, and lock programmer has to cope with problems.

- Two approaches: ad hoc and formal.
  - ad hoc: programmer manually augments all data races with pragmas to restrict compiler and hardware optimizations: not portable but often optimal.
  - formal: language has memory model and mechanisms to abstractly define races in program: portable but often baroque and not optimal.
- data access / compiler (C/C++): **volatile** qualifier
  - forces variable updates to occur quickly (at **sequence points**)
  - created for longjmp or force access for memory-mapped devices
  - for architectures with few registers, practically all variables are implicitly volatile. Why?
  - Java **volatile** / C++11 atomic stronger ⇒ prevent eliding ***and*** disjoint reordering.
- program order / compiler: disable inlining, **asm**("" ::: "memory");
- memory order / runtime (x86): sfence, lfence, mfence
  - guarantee previous stores and/or loads are completed, before continuing.
- atomic operations CAA/V, which often imply fencing
- platform-specific instructions for cache invalidation
- most mechanisms specific to platform or compiler
- difficult, low-level, diverse semantics, not portable ⇒ ***tread carefully!***
- Dekker for TSO:

```
#define CALIGN __attribute__(( aligned (64) ))  // cache-line alignment
#define Fence() __asm__ __volatile__ ( "mfence" ) // prevent hardware reordering
#define Pause() __asm__ __volatile__ ( "pause" : : : ) // efficient busy wait

enum Intent { DontWantIn, WantIn } Last;
_Task Dekker {
    volatile Intent &me, &you, *&Last;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            for ( ;; ) {                     // entry protocol
                me = WantIn;                 // high priority
                Fence();
              if ( you == DontWantIn ) break;
                if ( Last == &me ) {         // high priority ?
                    me = DontWantIn;
                    Fence();
                    while ( Last == &me ) Pause();  // low priority
                }
                Pause();
            }
```

```
                CriticalSection();          // critical section
                Last = &me;                 // exit protocol
                me = DontWantIn;
            }
        }
      public:
        Dekker( volatile Intent &me, volatile Intent &you, volatile Intent *&Last ) :
            me(me), you(you), Last(Last) {}
};
void uMain::main() {
        volatile Intent me CALIGN = DontWantIn, you CALIGN = DontWantIn,
                *Last CALIGN = rand() % 2 ? &me : &you;
        Dekker t0(me, you, Last), t1(you, me, Last);
};
```

- Locks using these features ensure SC for protected shared variables.
    - **no user races and strong locks ⇒ SC memory model**

11 Other Approaches

11.1 Exotic Atomic Instruction

- The compare-and-assign instruction performs an atomic compare and conditional assignment CAA (erroneously called compare-and-swap, CAS).

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

```
bool CAA( int &val,                  void Task::main() { // each task does
  int comp, int nval ) {                 while ( ! CAA(Lock,OPEN,CLOSED));
    // begin atomic                      // critical section
    if (val == comp) {                   Lock = OPEN;
        val = nval;                  }
        return true;
    }
    return false;
    // end atomic
}
```

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

- Alternative implementation assigns comparison value with the value when not equal.

```
bool CAV( int &val, int &comp, int nval ) {
    // begin atomic
    if (val == comp) {
        val = nval;
        return true;
    }
    comp = val;  // return changed value
    return false;
    // end atomic
}
```

- Assignment when unequal is useful as it also returns the new changed value.

- VAX computer has instructions to atomically insert and remove a node to/from the head or tail of a circular doubly linked list.

```
struct links {
    links *front, *back;
}
bool INSQUE( links &entry, links &pred ) {   // atomic execution
    // insert entry following pred
    return entry.front == entry.back;        // first node inserted ?
}
bool REMQUE( links &entry ) {                // atomic execution
    // remove entry
    return entry.front == null;              // last node removed ?
}
```

- MIPS processor has two instructions that generalize atomic read/write cycle: LL (load locked) and SC (store conditional).
  - LL instruction loads (reads) a value from memory into a register, and sets a hardware **reservation** on the memory from which the value is fetched.
  - Register value can be modified, even moved to another register.
  - SC instruction stores (writes) new value back to original or another memory location.
  - However, store is conditional and occurs only if no interrupt, exception, or write has occurred at LL reservation.
  - Failure indicated by setting the register containing the value to be stored to 0.
  - E.g., implement test-and-set with LL/SC:

```
int testSet( int &lock ) {        // atomic execution
    int temp = lock;              // read
    lock = 1;                     // write
    return temp;                  // return previous value
}
testSet:                          // register $4 contains pointer to lock
    ll   $2,($4)                  // read and lock location
    or   $8,$2,1                  // set register $8 to 1 (lock | 1)
    sc   $8,($4)                  // attempt to store 1 into lock
    beq $8,$0,testSet             // retry if interference between read and write
    j    $31                      // return previous value in register $2
```

  - Does not suffer from ABA problem.

```
Node *pop( Header &h ) {
    Node *t, next;
    for ( ;; ) {                          // busy wait
        t = LL( h.top );
      if ( t == NULL ) break;             // empty list ?
        next = t->next
      if ( SC( h.top, next ) ) break; // attempt to update top node
    }
    return t;
}
```

  - SC detects ***change*** to top rather than indirectly checking if value in top changes (CAA).
  - However, most architectures support weak LL/SC.
    - reservation granularity may be cache line or memory block rather than word
    - ⇒ SC fails but no write to LL address: false retry can result in incorrect result, e.g., fetch-and-add with LL/SC does another add
  - Cannot implement atomic swap as two reservations are necessary.

- Hardware transactional memory allows up to N (4,6,8) reservations, e.g, Advanced Synchronization Facility (ASF) proposal in AMD64.

- Like database **transaction** that optimistically executes change, and either commits changes, or rolls back and restarts if interference.

- SPECULATE : start speculative region and clear zero flag; next instruction checks for abort and branches to retry.
- LOCK : MOV instruction indicates location for atomic access, but moves not visible to other CPUs.
- COMMIT : end speculative region
  - if no conflict, make MOVs visible to other CPUs.
  - if conflict to any move locations, set zero flag, discard reservations and restore registers back to instruction following SPECULATE
- Can implement several data structures without ABA problem.

- Software Transactional Memory (STM) : allow any number of reservations.
  - atomic blocks of arbitrary size:

```
void push( header &h, node &n ) {
    atomic {           // SPECULATE
        n.next = h.top;
        h.top = &n
    }                  // COMMIT
}
```

  - implementation records all memory locations read and written, and all values mutated.
    - bookkeeping costs and rollbacks typically result in significant performance degradation
  - alternative implementation inserts locks throughout program to protect shared access
    - finding all access is difficult and ordering lock acquisition is complex

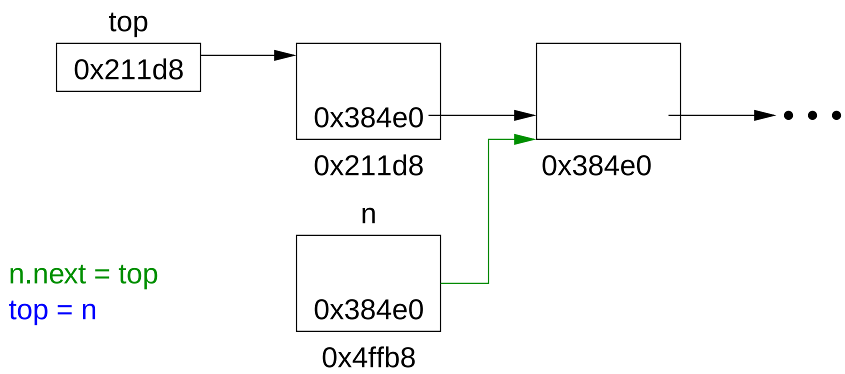
11.2 Atomic (Lock-Free) Data-Structure

- Called **lock free** if data-structure operations that are critical sections can be performed without a lock.
  - e.g., adding or removing a node to a data structure without using a lock.
- If operations also guarantees progress, called **wait free**.
- E.g., build a stack with lock-free push and pop operations.

```
struct Node {
    // data
    Node *next; // pointer to next node
};
struct Header {
    Node *top;  // pointer to stack top
};
```

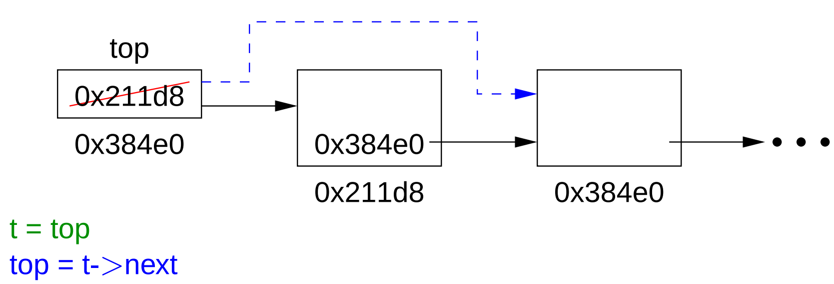
- Use CAA to atomically update top pointer when nodes pushed or popped concurrently.

```
void push( Header &h, Node &n ) {
    for ( ;; ) {                          // busy wait
        n.next = h.top;                   // link new node to top node
      if ( CAA( h.top, n.next, &n ) ) break; // attempt to update top node
    }
}
```



- Create new node, n, at 0x4ffb8 to be added.
- Set n.next to h.top.
- CAA tries to assign new top &n to h.top.
- CAA fails if top changed since copied to n.next
- If CAA failed, update n.next to h.top, and try again.
- CAA succeeds when top == n.next, i.e., no push or pop between setting n.next and trying to assign &n to h.top.
- CAV copies changed value to n.next, so eliminates reseting t = h.top in busy loop.

```
Node *pop( Header &h ) {
    Node *t, *next;
    for ( ;; ) {                          // busy wait
        t = h.top;                        // copy current top
      if ( t == NULL ) break;             // empty list ?
        next = t->next                    // copy new top
      if ( CAA( h.top, t, next ) ) break; // attempt to update top node
    }
    return t;
}
```



- Copy top node, 0x4ffb8, to t for removal.
- If not empty, attempt CAA to set new top to next node, t->next.
- CAA fails if top changed since copied to t.
- If CAA failed, update t to h.top, and try again.
- CAA succeeds when top == t->next, i.e., no push or pop between setting t and trying to assign t->next to h.top.
- CAV copies the changed value into t, so eliminates reseting t = h.top in busy loop.

11.2.1 ABA problem

- Pathological failure for a particular series of pops and pushes, called **ABA problem**.
- Given stack with 3 nodes:

  top → x → y → z

- Popping task, T_i, has t set to x and dereferenced t->next to get the local value of y for its argument to CAA.
- T_i is now time-sliced **before the CAA**, and while blocked, nodes x and y are popped, and x is pushed again:

  top → x → z

- When T_i restarts, CAA successfully removes x as same header before time-slice.
- ***But now incorrectly sets h.top to its local value of node y:***

  top → y → ???

  stack is now corrupted!!!

11.2.2 Hardware Fix

- Probabilistic solution for stack exists using double-wide CAVD instruction, which compares and assigns 64-bit (128-bit) values.

```
bool CAVD( uint64_t &val, uint64_t &comp, uint64_t nval ) {
    /* begin atomic */
    if ( val == comp ) {        // 64-bit compare
        val = nval;             // 64-bit assignment
        return true;
    }
    comp = val;                 // 64-bit assignment
    return false;
    /* end atomic */
}
```

- Now, associate counter (ticket) with header node:

```
struct Header {               // 64-bit (2 x 32-bit) or 128-bit (2 x 64-bit)
    Node *top;                // pointer to stack top
    int count;
};
struct Node {
    // data
    Header next;              // pointer to next node / count
};
```

- Increment counter in push so pop can detect ABA if node re-pushed.

```
void push( Header &h, Node &n ) {
    CAVD( n.next, n.next, h );   // atomic assignment: n.next = h
    for ( ;; ) {                 // busy loop
      if ( CAVD( h, n.next, (Header){ &n, n.next.count + 1 } ) ) break;
    }
}
```

  - CAVD used to copy entire header to n.next, as structure assignment (2 fields) is not atomic.
  - In busy loop, copy local idea of top to next of new node to be added.
  - CAVD tries to assign new top-header to (h).
  - If top has not changed since copied to n.next, update h.top to n (new top), and ***increment counter***.
  - If top has changed, CAVD copies changed values to n.next, so try again.

```
Node *pop( Header &h ) {
    Header t;
    CAVD( t, t, h );              // atomic assignment: t = h
    for ( ;; ) {                  // busy loop
      if ( t.top == NULL ) break;   // empty list ?
      if ( CAVD( h, t, (Header){ t.top->next.top, t.count } ) ) break;
    }
    return t.top;
}
```

  - CAVD used to copy entire header to t, as structure assignment (2 fields) is not atomic.
  - In busy loop, check if pop on empty stack and return NULL.
  - If not empty, CAVD tries to assign new top t.top->next.top,t.count to h.
  - If top has not changed since copied to t, update h.top to t.top->next.top (new top).
  - If top has changed, CAVD copies changed values to t, so try again.

- ABA problem (mostly) fixed:

  top,3 → x → y → z

- Popping task, T_i, has t set to x,3 and dereferenced y from t.top->next in argument of CAVD.
- T_i is time-sliced, and while blocked, nodes x and y are popped, and x is pushed again:

  top,4 $\rightarrow$ x $\rightarrow$ z   *// adding x increments counter*

- When T_i restarts, CAVD fails as header x,3 not equal top x,4.
- Only probabilistic correct as counter finite (like ticket counter).
  - task T_i is time-sliceed and sufficient pushes wrap counter to value stored in T_i's header,
  - node x just happens to be at the top of the stack when T_i unblocks.
- Doubtful if failure can arise, given 32/64-bit counter and pathological case.
- Finally, none of the programs using CAA ensure eventual progress; therefore, rule 5 is broken.

11.2.3 Hardware/Software Fix

- Fixing ABA with CAA/V and more code is extremely complex (100s of lines of code), as is implementing more complex data structures (queue, deque, hash).
- All solutions require complex determination of when a node has no references (like garbage collection).
  - each thread maintains a list of accessed nodes, called **hazard pointers**
  - thread updates its hazard pointers while other threads are reading them
  - thread removes a node by hiding it on a private list and periodically scans the hazard lists of other threads for referencing to that node
  - if no pointers are found, the node can be freed
- For lock-free stack: x, y, z are memory addresses
  - first thread puts 'x' on its hazard list
  - second thread cannot reuse 'x', because of hazard list
  - second thread must create new object at different location
  - first thread detects change
- Summary: locks versus lock-free
  - lock-free has no lock, cannot contribute to deadlock
  - when thread blocks, it is never holding a lock to prevent progress of other threads
  - lock-free no panacea, performance unclear
  - lock can protect arbitrarily complex critical section
    vs. lock-free can only handle limited set of critical sections
  - combine lock and lock-free?

11.3 Concurrency Languages

11.3.1 Ada 95

- Like μC++, but not as general.
- E.g., monitor bounded-buffer, restricted implicit (automatic) signal:

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

- The **when** clause is 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 direct dating-service.
- In contrast, if local variables are allowed, dating service is solvable.

```
_Monitor DatingService {
    int girls[noOfCodes], boys[noOfCodes];   // count girls/boys waiting
    bool exchange;                           // performing phone-number exchange
    int girlPhoneNo, boyPhoneNo;             // communication variables
  public:
    int girl( int phoneNo, int ccode ) {
        girls[ccode] += 1;
        if ( boys[ccode] == 0 ) {            // no boy waiting ?
            WAITUNTIL( boys[ccode] != 0, , ); // use parameter, not at start
            boys[ccode] -= 1;                // decrement dating pair
            girls[ccode] -= 1;
            girlPhoneNo = phoneNo;           // girl's phone number for exchange
            exchange = false;                // wake boy
        } else {
            girlPhoneNo = phoneNo;           // girl's phone number before exchange
            exchange = true;                 // start exchange
            WAITUNTIL( ! exchange, , );      // wait until exchange complete, not at start
        }
        RETURN( boyPhoneNo );
    }
    // boy
};
```

- 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 only appear in **task** main.
- Hence, Ada has no direct internal-scheduling mechanism, i.e., no condition variables.
- Instead a **requeue** statement can be used to make a ***blocking*** call to another (usually non-public) mutex member of the object.
- The original call is re-blocked on that mutex member's entry queue, which can be subsequently accepted when it is approriate to restart it.
- However, all **requeue** techniques suffer the problem of dealing with accumulated temporary results:
  - If a call must be postponed, its temporary results must be returned and bundled with the initial parameters before forwarding to the mutex member handling the next step,
  - or the temporary results must be re-computed at the next step (if possible).
- In contrast, waiting on a condition variable automatically saves the execution location and any partially computed state.
- Can also use **requeue** with Ada monitor and arrays of mutex routines:

```
generic
    type ccset is range<>;                    -- template parameter
package DatingServicePkg is                   -- interface
    type TriggerType is array(ccset) of Boolean;

    protected type DatingService is           -- interface
        entry Girl( Partner : out Integer; PhNo : in Integer; ccode : in ccset );
        entry Boy( Partner : out Integer; PhNo : in Integer; ccode : in ccset );
      private
        entry Girls(ccset)( Partner : out Integer; PhNo : in Integer; ccode : in ccset );
        entry Boys(ccset)( Partner : out Integer; PhNo : in Integer; ccode : in ccset );
        entry Exchange( Partner : out Integer; PhNo : in Integer; ccode : in ccset );

        ExPhNo, GPhNo, BPhNo : Integer;
        GTrig, BTrig : TriggerType := (ccset => false); -- initialize array to false
        ExTrig : Boolean := false;
    end DatingService;
end DatingServicePkg;

package body DatingServicePkg is              -- implementation
    protected body DatingService is
    entry Girl( Partner : out Integer;        -- mutex routine
            PhNo : in Integer; ccode : in ccset )
      when exchange'count = 0 is              -- automatic signal
    begin
        if Boys(ccode)'count = 0 then         -- size of mutex queue
            requeue Girls(ccode);             -- call postponed
        else
            GPhNo := PhNo; BTrig(ccode) := true;
            requeue exchange;                 -- call postponed
        end if;
    end Girl;
    -- boy code is similar

    entry Girls(for code in ccset)( Partner : out Integer;
            PhNo : in Integer; ccode : in ccset )
      when GTrig(code) is
    begin
        GTrig(ccode) := false;  ExPhNo := PhNo;
        Partner := BPhNo;  ExTrig := true;
    end Girls;
    -- boy code is similar

    entry exchange( Partner : out Integer;
            PhNo : in Integer; ccode : in ccset )
      when ExTrig is
    begin
        ExTrig := false;  Partner := ExPhNo;
    end Exchange;
```

- Generic in the number of compatibility codes.
- Array of Girls and Boys entry routines (entry family).

- ○ Requeue on entry family if compatible partner is unavailable.
- ○ Use array of boolean triggers to indicate which members can run.
- ○ Do not allow public calls if exchange is occurring.

11.3.2 SR/Concurrent C++

- SR and Concurrent C++ have tasks with external scheduling using an accept statement.
- But no condition variables or requeue statement.
- To ameliorate internal scheduling there is a **when** and by clause on the **accept** statement.
- **when** clause is allowed to reference caller's arguments via parameters of mutex member:

```
select
    accept mem( code : in Integer )
        when code % 2 = 0 do ...    -- accept call with even code
or
    accept mem( code : in Integer )
        when code % 2 = 1 do ...    -- accept call with odd code
end select;
```

- Placement of **when** clause after the **accept** clause so parameter names are defined.
- **when** referencing parameter ⇒ implicit search of waiting tasks on mutex queue ⇒ locking mutex queue.
- Select longest waiting if multiple true **when** clauses.
- **by** clause is calculated for each true **when** clause and the minimum **by** clause is selected.

```
select
    accept mem( code : in Integer )
        when code % 2 = 0 by -code do ... -- accept call with largest even code
or
    accept mem( code : in Integer )
        when code % 2 = 1 by code do ... -- accept call with smallest odd code
end select;
```

- Select longest waiting if multiple by clauses with same minimum.
- **by** clause exacerbates the execution cost of computing an **accept** clause.
- While **when**/**by** remove some internal scheduling and/or requeue, constructing expressions can be complex.
- There are still valid situations neither can deal with, e.g., if the selection criteria involves multiple parameters:
  - ○ select the lowest even value of code1 and the highest odd value of code2 if there are multiple lowest even values.

  ◦ selection criteria involves information from other mutex queues such as the dating service (girl must search the boy mutex queue).

- Often simplest to unconditionally accept a call allowing arbitrarily examination, and possibly postpone (internal scheduling).

11.3.3 Java

- Java's concurrency constructs are largely derived from Modula-3.

```
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 μC++ uBaseTask, and all tasks must explicitly inherit from it:

```
class myTask extends Thread { // inheritance
    private int arg;          // communication variables
    private int result;
    public mytask() {...}     // task constructors
    public void run() {...}   // task main
    public int result() {...} // return result
    // unusual to have more members
}
```

- Thread starts in member run.
- Java requires explicit starting of a thread by calling start after the thread's declaration.

  ⇒ coding convention to start thread or inheritance is precluded (can only start a thread once)

- Termination synchronization is accomplished by calling join.
- Returning a result on thread termination is accomplished by member(s) returning values from the task's global variables.

```
mytask th = new myTask(...); // create and initialized task
th.start();           // start thread
// concurrency
th.join();            // wait for thread termination
a2 = th.result();     // retrieve answer from task object
```

- Like μC++, when the task's thread terminates, it becomes an object, hence allowing the call to result to retrieve a result.
- (see Section 8.8, p. 141 for monitors)

11.3.4 Go

- Light-weight (like μC++) ***non-preemptive?*** threads (called **goroutine**).
  - ⇒ busy waiting only on multicore (Why?)
- **go** statement (like start/fork) creates new user thread running in routine.

```
go foo( 3, f )    // start thread in routine foo
```

- Arguments may be passed to goroutine but return value is discarded.
- **Cannot reference goroutine object** ⇒ no direct communication.
- All threads terminate silently when program terminates.
- Threads synchronize/communicate via **channel** (CSP) ⇒ **paradigm shift from routine call**.
- Channel is a typed shared buffer with 0 to N elements.

```
ch1 := make( chan int, 100 )      // integer channel with buffer size 100
ch2 := make( chan string )        // string channel with buffer size 0
ch2 := make( chan chan string ) // channel of channel of strings
```

- Buffer size > 0 ⇒ up to N asynchronous calls; otherwise, synchronous call.
- Operator `<-` performs send/receive; send: `ch1 <- 1`, receive: `s <- ch2`
- Channel can be constrained to only send or receive; otherwise bi-directional.

```go
package main
import "fmt"
func main() {

    type Msg struct{ i, j int }
    ch1 := make( chan int )
    ch2 := make( chan float32 )
    ch3 := make( chan Msg )
    hand := make( chan string )
    shake := make( chan string )
    gortn := func() {
        var i int;  var f float32;  var m Msg
        L: for {
            select { // wait for message
              case <- hand: break L // sentinel
              case i = <- ch1: fmt.Println( i )
              case f = <- ch2: fmt.Println( f )
              case m = <- ch3: fmt.Println( m )

            }
        }
        shake <- "SHAKE" // completion
    }

    go gortn()          // start thread in gortn
    ch1 <- 0            // different messages
    ch2 <- 2.5
    ch3 <- Msg{1, 2}
    hand <- "HAND" // sentinel value
    <-shake             // wait for completion
}
```

```cpp
#include <iostream>
using namespace std;
_Task Gortn {
  public:
    struct Msg { int i, j; };
    void mem1( int i ) { Gortn::i = i; }
    void mem2( float f ) { Gortn::f = f; }
    void mem3( Msg m ) { Gortn::m = m; }
  private:
    int i;  float f;  Msg m;
    void main() {

        for ( ;; ) {

            _Accept( ~Gortn ) break;
            or _Accept( mem1 ) cout << i << endl;
            or _Accept( mem2 ) cout << f << endl;
            or _Accept( mem3 ) cout << "{" << m.i
                    << " " << m.j << "}" << endl;
        }
    }

};
void uMain::main() {
    Gortn gortn;
    gortn.mem1( 0 );
    gortn.mem2( 2.5 );
    gortn.mem3( (Gortn::Msg){ 1, 2 } );

} // wait for completion
```

- Locks

```go
type Cond                  // synchronization lock
    func NewCond(l Locker) *Cond
    func (c *Cond) Broadcast()
    func (c *Cond) Signal()
    func (c *Cond) Wait()
type Mutex                 // mutual exclusion lock
    func (m *Mutex) Lock()
    func (m *Mutex) Unlock()
type Once                  // singleton-pattern
    func (o *Once) Do(f func())
type RWMutex               // readers/writer lock
    func (rw *RWMutex) Lock()
    func (rw *RWMutex) RLock()
    func (rw *RWMutex) RLocker() Locker
    func (rw *RWMutex) RUnlock()
    func (rw *RWMutex) Unlock()
type WaitGroup             // countdown lock
    func (wg *WaitGroup) Add(delta int)
    func (wg *WaitGroup) Done()
    func (wg *WaitGroup) Wait()
```

- Atomic operations

```
func AddInt32(val *int32, delta int32) (new int32)
func AddInt64(val *int64, delta int64) (new int64)
func AddUint32(val *uint32, delta uint32) (new uint32)
func AddUint64(val *uint64, delta uint64) (new uint64)
func AddUintptr(val *uintptr, delta uintptr) (new uintptr)
func CompareAndSwapInt32(val *int32, old, new int32) (swapped bool)
func CompareAndSwapInt64(val *int64, old, new int64) (swapped bool)
func CompareAndSwapPointer(val *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)
func CompareAndSwapUint32(val *uint32, old, new uint32) (swapped bool)
func CompareAndSwapUint64(val *uint64, old, new uint64) (swapped bool)
func CompareAndSwapUintptr(val *uintptr, old, new uintptr) (swapped bool)
func LoadInt32(addr *int32) (val int32)
func LoadInt64(addr *int64) (val int64)
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
func LoadUint32(addr *uint32) (val uint32)
func LoadUint64(addr *uint64) (val uint64)
func LoadUintptr(addr *uintptr) (val uintptr)
func StoreInt32(addr *int32, val int32)
func StoreInt64(addr *int64, val int64)
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
func StoreUint32(addr *uint32, val uint32)
func StoreUint64(addr *uint64, val uint64)
func StoreUintptr(addr *uintptr, val uintptr)
```

11.3.5 C++11 Concurrency

- C++11 library may be sound because C++ now has a stronger memory-model.

- compile: `g++ -std=c++11 -lpthread ...`

- Thread creation: start/wait (fork/join) approach.

```
class thread {
  public:
    template <class Fn, class... Args>
        explicit thread( Fn &&fn, Args &&... args );
    void join();                        // termination synchronization
    bool joinable() const;              // true => joined, false otherwise
    void detach();                      // independent lifetime
    id get_id() const;                  // thread id
};
```

- Any entity that is *callable* (functor) may be started:

```
#include <thread>
void hello( const string &s ) {          // callable
    cout << "Hello " << s << endl;
}
class Hello {                            // functor
    int result;
  public:
    void operator()( const string &s ) { // callable
        cout << "Hello " << s << endl;
    }
};

int main() {
    thread t1( hello, "Peter" );         // start thread in routine "hello"
    Hello h;                             // thread object
    thread t2( h, "Mary" );              // start thread in functor "h"
    // work concurrently
    t1.join();                           // termination synchronization
    // work concurrently
    t2.join();                           // termination synchronization
} // must join before closing block
```

- Passing multiple arguments uses C++11's variadic template feature to provide a type-safe call chain via thread constructor to the *callable* routine.
- Thread starts implicit at point of declaration.
- Instead of join, thread can run independently by detaching:

```
t1.detach();    // "t1" must terminate for program to end
```

- Beware dangling pointers to local variables:

```
{
    string s( "Fred" );         // local variable
    thread t( hello, s );
    t.detach();
} // "s" deallocated and "t" running with reference to "s"
```

- **It is an error to deallocate thread object before join or detach.**
- Locks
  - mutex, recursive, timed, recursive-timed

```
class mutex {
  public:
    void lock();                // acquire lock
    void unlock();              // release lock
    bool try_lock();            // nonblocking acquire
};
```

- condition

```
class condition_variable {
  public:
    void notify_one();           // unblock one
    void notify_all();           // unblock all
    void wait( mutex &lock );    // atomically block & release lock
};
```

- Scheduling is no-priority nonblocking ⇒ barging ⇒ wait statements must be in while loops to recheck conditions.

```
#include <mutex>
class BoundedBuffer {              // simulate monitor
    // buffer declarations
    mutex mlock;                   // monitor lock
    condition_variable empty, full;
    void insert( int elem ) {
        mlock.lock();
        while (count == Size ) empty.wait( mlock );   // release lock
        // add to buffer
        count += 1;
        full.notify_one();
        mlock.unlock();
    }

    int remove() {
        mlock.lock();
        while( count == 0 ) notempty.wait( mlock );   // release lock
        // remove from buffer
        count -= 1;
        empty.notify_one();
        mlock.unlock();
        return elem;
    }
};
```

- Futures

```
#include <future>
big_num pi( int decimal_places ) {...}
int main() {
    future<big_num> PI = async( pi, 1200 ); // PI to 1200 decimal places
    // work concurrently
    cout << "PI " << PI.get() << endl;        // block for answer
}
```

- Atomic types/operations

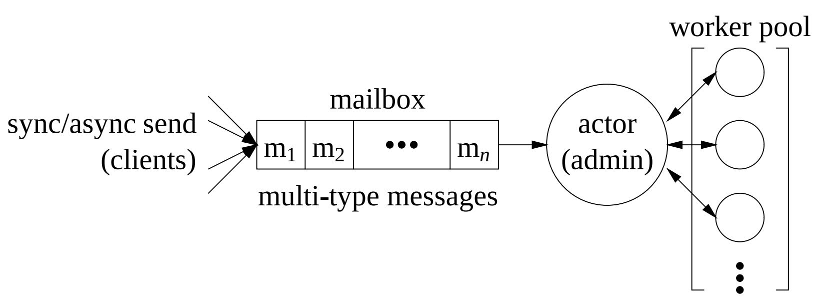
atomic_flag, atomic_bool, atomic_char, atomic_schar, atomic_uchar, atomic_short, atomic_ushort, atomic_int, atomic_uint, atomic_long, atomic_ulong, atomic_llong, atomic_ullong, atomic_wchar_t, atomic_address, atomic<T>

```
typedef struct atomic_itype {
    bool operator=(int-type) volatile;
    void store(int-type) volatile;
    int-type load() const volatile;
    int-type exchange(int-type) volatile;
    bool compare_exchange(int-type &old_value, int-type new_value) volatile;
    int-type fetch_add(int-type) volatile;
    int-type fetch_sub(int-type) volatile;
    int-type fetch_and(int-type) volatile;
    int-type fetch_or(int-type) volatile;
    int-type fetch_xor(int-type) volatile;

    int-type operator++() volatile;
    int-type operator++(int) volatile;
    int-type operator--() volatile;
    int-type operator--(int) volatile;
    int-type operator+=(int-type) volatile;
    int-type operator-=(int-type) volatile;
    int-type operator&=(int-type) volatile;
    int-type operator|=(int-type) volatile;
    int-type operator^=(int-type) volatile;
} atomic_itype;
```

11.4 Concurrency Models

11.4.1 Actors

- An **actor** (Hewitt/Agha) is an administrator, accepting messages, handles it, or sends it to a worker in a fixed or open pool.
- However, communication is via an untyped queue of messages.



- Mmailbox is polymorphic in message types ⇒ dynamic type-checking.
- Messages send is usually asynchronous.
- Two popular programming languages using actors are Erlang and Scala.

11.4.1.1 Scala (2.10)

- Scala is a statically-typed object-oriented/functional language (version dependent, both syntax and semantics).

- Monitor (Java style), no-priority nonblocking ⇒ barging

```
class BoundedBuffer[T]( Size: Int ) {
    var count = 0
    val elems = new Queue[T]
    def put( elem: T ) = synchronized {
        while ( count == Size ) wait()
        elems.enqueue( elem )
        count += 1
        if ( count == 1 ) notifyAll()
    }
    def get: T = synchronized {
        while ( count == 0 ) wait()
        val elem = elems.dequeue
        count -= 1
        if ( count == Size - 1 ) notifyAll()
        return elem
    }
}
```

- Thread created via **actor**: a task with a public atomic message-queue.

```
object ProdCons {
    def main( args: Array[String] ) {       // main program
        val buf = new BoundedBuffer[Int](10)
        val prod = new Actor {
            def act() {                     // thread starts here
                for ( i <- 0 until 10 ) { buf.put( i ) }
                buf.put( -1 )               // sentinel
            }
        }
        prod.start                          // explicit start
        val cons = new Actor {
            def act() {                     // thread starts here
                var v = 0;
                breakable { while ( true ) {
                    v = buf.get;
                  if ( v == -1 ) break;     // sentinel ?
                    println( v );
                } }
            }
        }
        cons.start                          // explicit start
    }
}
```

- Communication is via an untyped queue of messages.

- Caller sends message synchronously, asynchronously, or asynchronously-with-future.

```
val actor = new Actor { ... }
val v = actor !? 'c'        // synchronous send, wait for result
actor ! ( 1, 3.1 )          // asynchronous send, no result
val f = actor !! "Hi"       // asynchronous send, result in future
```

- Scala actor can conditionally receive messages from mailbox.

```
class Administrator extends Actor {
    case object Stop                      // local type
    def act() {                           // thread starts here
        while ( true ) {
            receive {
              case c: Char => ...         // char message
              case ( i: Int, d: Double ) => ...  // tuple message
              case s: String => ...       // String message
              case Stop => ...            // Stop message
              case _ => ...               // unknown message
            }
        }
    }
}

val admin = new Administrator
admin.start
val v = admin !? 'c'                      // synchronous send
admin ! ( 1, 3.1 )                        // asynchronous send
admin ! ( 2, 3.2 )                        // asynchronous send
val f = admin !! "Hi"                     // asynchronous send with future
... // do some work concurrently
val result = f()                          // block for result
admin !? admin.Stop                       // special stop message
```

- Difference between Scala and μC++
  - dynamic versus static communication
  - match message type versus member routine
  - GC allows detached threads whereas μC++ must join
  - asynchronous send versus synchronous call

11.4.2 Linda

- Based on a multiset **tuple space**, duplicates are allowed, accessed associatively by threads for synchronization and communication.

```
( 1 )               // 1 element tuple of int
( 1.0, 2.0, 3.0 )   // 3 element tuple of double
( 'M', 5.5 )        // 2 element tuple of char, double
```

- Often there is only one tuple space implicitly known to all threads ⇒ tuple space is unnamed.

- Threads created through tuple space, and communicate by adding, reading and removing data from tuple space.
- A tuple is accessed atomically and by content, where content is based on the number of tuple elements, the data types of the elements, and possibly their values.
- At least 3 operations are available for accessing the tuple space:

  1. The blocking read operation reads a tuple from the tuple space:

     ```
     read( 'M', 34, 5.5 );
     ```

     blocks until a matching 3 field tuple with values ('M', 34, 5.5) appears in tuple space.

     A general form is possible using a type rather than a value, e.g.:

     ```
     int i;
     read( 'M', ?i, 5.5 );
     ```

     blocks until a matching 3 field tuple with values ('M', **int**, 5.5), where **int** matches any integer value, and value copied to i.
  2. The blocking in operation is identical to read operation, but removes the matching tuple from the tuple space.
  3. The non-blocking out operation writes a tuple to the tuple space, e.g.:

     ```
     out( 'M', 34, 5.5 );
     ```

     A variable can be used instead of a constant.

- Semaphore:

Mutual Exclusion	Synchronization	
	Task$_1$	Task$_2$

out("semaphore"); // open
...
in("semaphore");
// critical section
out("semaphore");
	in("semaphore");
	S2
		S1
		out("semaphore");

- Producer/Consumer with bounded buffer:

```
for ( i = 0; i < bufSize; i += 1 ) {            // number of buffer elements
    out( "bufferSem" );
}
void producer() {
    for ( int i = 1; i <= 20; i += 1 ) {
        in( "bufferSem" );                      // wait for empty buffer slot
        out( "buffer", rand() % 100 );          // insert element into tuple space
    }
    in( "bufferSem" );                          // wait for empty buffer slot
    out( "buffer", -1 );                        // insert element into tuple space
}
```

```
int consumer() {
    int elem;
    for ( ;; ) {
        in( "buffer", ?elem );        // remove element from tuple space
        out( "bufferSem" );           // indicate empty buffer slot
      if ( elem == -1 ) break;
        // consume element
    }
}
```

- Handles multiple consumers and producers because insertion and removal of elements from the tuple space is atomic.

11.4.3 OpenMP

- Shared memory, implicit thread management (programmer hints), 1-to-1 threading model (kernel threads), some explicit locking.
- Communicate with compiler with #pragma directives.

```
#pragma omp ...
```

- fork/join model
  - fork: initial thread creates a team of parallel threads (including itself)
  - each thread executes the statements in the region construct
  - join: when team threads complete, synchronize and terminate, except initial thread which continues
- COBEGIN/COEND: each thread executes different section:

```
#include <omp.h>
... // declarations of p1, p2, p3
int main() {
    int i;
    #pragma omp parallel sections num_threads( 4 ) // fork "rows" thread-team
    { // COBEGIN
        #pragma omp section
        { i = 1; }
        #pragma omp section
        { p1( 5 ); }
        #pragma omp section
        { p2( 7 ); }
        #pragma omp section
        { p3( 9 ); }
    } // COEND (synchronize)
}
```

- compile: `gcc -Wall -std=c99 -fopenmp openmp.c -lgomp`

- All threads execute same section:

```
int main() {
    const unsigned int rows = 10, cols = 10;    // sequential
    int matrix[rows][cols], subtotals[rows], total = 0;

    // read matrix
    #pragma omp parallel num_threads( rows )  // fork "rows" thread-team
    {                                          // concurrent
        unsigned int row = omp_get_thread_num(); // row to add is thread ID
        subtotals[row] = 0;
        for ( unsigned int c = 0; c < cols; c += 1 ) {
            subtotals[row] += matrix[row][c];
        }
    }                                          // join thread-team
    for ( unsigned int r = 0; r < rows; r += 1 ) {  // sequential
        total += subtotals[r];
    }
    printf( "total:%d\n", total );
} // main
```

- Variables outside section are shared; variables inside section are thread private.
- **for** directive specifies each loop iteration is executed by a team thread

```
#pragma omp parallel for                       // fork "rows" thread-team
for ( unsigned int r = 0; r < rows; r += 1 ) {  // concurrent
    subtotals[r] = 0;
    for ( unsigned int c = 0; c < cols; c += 1 ) {
        subtotals[r] += matrix[r][c];
    }
}
```

- In this case, sequential code directly converted to concurrent via #pragma.
- Programmer responsible for any overlapping sharing in vector/matrix manipulation.
- barrier (also critical section and atomic directives)

```
int main() {
    #pragma omp parallel num_threads( 4 )       // fork "rows" thread-team
    {
        sleep( omp_get_thread_num() );
        printf( "%d\n", omp_get_thread_num() );
        #pragma omp barrier                    // wait for all threads to finish
        printf("sync\n");
    }
}
```

11.5 Threads & Locks Library

11.5.1 java.util.concurrent

- Java library is sound because of memory-model and language is concurrent aware.

- Synchronizers : Semaphore (counting), CountDownLatch, CyclicBarrier, Exchanger, Condition, Lock, ReadWriteLock

- Use new locks to build a monitor with multiple condition variables.

```
class BoundedBuffer {                          // simulate monitor
    // buffer declarations
    final Lock mlock = new ReentrantLock();     // monitor lock
    final Condition empty  = mlock.newCondition();
    final Condition full = mlock.newCondition();
    public void insert( Object elem ) throws InterruptedException {
        mlock.lock();
        try {
            while (count == Size ) empty.await();  // release lock
            // add to buffer
            count += 1;
            full.signal();
        } finally { mlock.unlock(); } // ensure monitor lock is unlocked
    }
    public Object remove() throws InterruptedException {
        mlock.lock();
        try {
            while( count == 0 ) full.await();  // release lock
            // remove from buffer
            count -= 1;
            empty.signal();
            return elem;
        } finally { mlock.unlock(); } // ensure monitor lock is unlocked
    }
}
```

  - Condition is nested class within ReentrantLock ⇒ condition implicitly knows its associated (monitor) lock.
  - Scheduling is still no-priority nonblocking ⇒ barging ⇒ wait statements must be in while loops to recheck condition.
  - No connection with implicit condition variable of an object.
  - **Do not mix implicit and explicit condition variables.**

- Executor/Future :
  - Executor is a server with one or more worker tasks (worker pool).
  - Call to executor submit is asynchronous and returns a future.
  - Future is closure with work for executor (Callable) and place for result.
  - Result is retrieved using get routine, which may block until result inserted by executor.

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;
public class Client {
    public static void main( String[] args )
            throws InterruptedException, ExecutionException {
        Callable<Integer> work = new Callable<Integer>() {
            public Integer call() {
                // do some work
                return d;
            }
        };
        ExecutorService executor = Executors.newFixedThreadPool( 5 );
        List<Future<Integer>> futures = new ArrayList<Future<Integer>>();
        for ( int f = 0; f < 10; f += 1 )
            // pass work to executor and store returned futures
            futures.add( executor.submit( work ) );
        for ( int f = 0; f < 10; f += 1 )
            System.out.println( futures.get( f ).get() );  // retrieve result
        executor.shutdown();
    }
}
```

- μC++ also has fixed thread-pool executor.

```
struct Work {                 // routine or functor
    int operator()() {
        // do some work
        return d;
    }
} work;
void uMain::main() {
    uExecutor executor( 5 );
    Future_ISM<int> futures[10];
    for ( unsigned int f = 0; f < 10; f += 1 )
        // pass work to executor and store returned futures
        executor.submit( futures[f], work );
    for ( unsigned int f = 0; f < 10; f += 1 )
        cout << futures[f]() << endl;  // retrieve result
}
```

- Collections : LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue, DelayQueue, ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet.
    - Create threads that interact indirectly through atomic data structures, e.g., producer/-consumer interact via LinkedBlockingQueue.
- Atomic Types using compare-and-assign (see Section **??**, p. ??) (i.e., lock-free).

  AtomicBoolean, AtomicInteger, AtomicIntegerArray, AtomicLong, AtomicLongArray, AtomicReference<V>, AtomicReferenceArray<E>

```
int v;                                                    | 1
AtomicInteger i = new AtomicInteger();                    | 2 2
i.set( 1 );                                               | 1 1
System.out.println( i.get() );                            | 2 1
v = i.addAndGet( 1 );            // i += delta            | 1 2
System.out.println( i.get() + " " + v );                  |
v = i.decrementAndGet();         // --i                   |
System.out.println( i.get() + " " + v );                  |
v = i.getAndAdd( 1 );            // i =+ delta            |
System.out.println( i.get() + " " + v );                  |
v = i.getAndDecrement();         // i--                   |
System.out.println( i.get() + " " + v );                  |
```

11.5.2 Pthreads

- Several libraries exist for C (pthreads) and C++ (μC++).
- C libraries built around routine abstraction and mutex/condition locks ("attribute" parameters not shown).

```
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.

```
void *rtn( void *arg ) { ... }
int i = 3, r, rc;
pthread_t t;              // thread id
rc = pthread_create( &t, rtn, (void *)i ); // create and initialized task
if ( rc != 0 ) ...        // check for error
// concurrency
rc = pthread_join( t, &r ); // wait for thread termination and result
if ( rc != 0 ) ...        // check for error
```

- All C library approaches have type-unsafe communication with tasks.
- No external scheduling ⇒ complex direct-communication emulation.
- Internal scheduling is no-priority nonblocking ⇒ barging ⇒ wait statements must be in while loops to recheck conditions

```
typedef struct {                        // simulate monitor
    // 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.

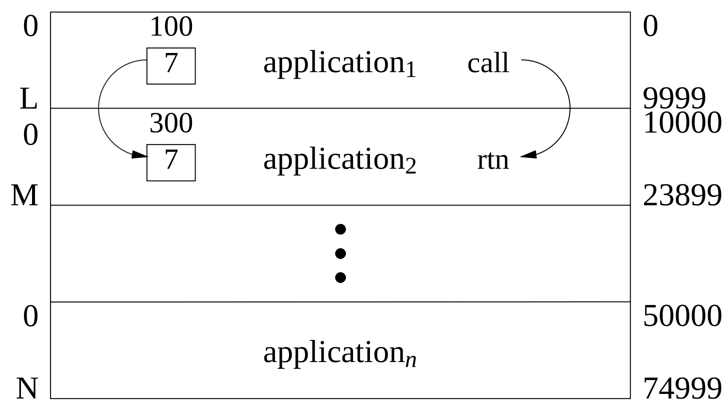
- Condition locks should only be accessed with mutual exclusion.
- pthread_cond_wait atomically blocks thread and releases mutex lock, which is necessary to close race condition on baton passing.

12 Distributed Environment

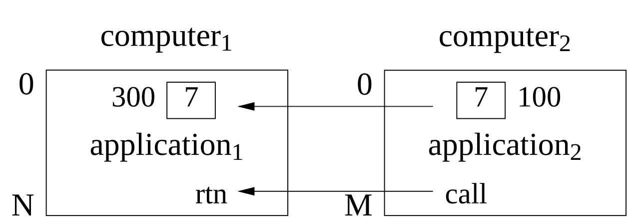
12.1 Multiple Address-Spaces

- An application executes in an **address space** (AS), which has its own set of addresses from 0 to N.

- Most computers can divide physical memory into multiple independent ASs (virtual memory).



- Often a process has its own AS, while a task does not (it exists inside a process's AS).
- Hence, multiple ASs are the beginnings of a distributed system as pointers no longer work.
- Another computer's memory is clearly another AS.



- The ability to call a routine in another address space is an important capability.
- Concurrency is implied by remote calls, but implicit, because each AS (usually) has its own thread of control and each computer has its own CPU.
- Communication is synchronous because the remote call's semantics are the same as a local call.

- Many difficult problems:

  1. How does the code for the called routine get on the other computer? Either it already exists on the other machine or it is shipped there dynamically (Java).
  2. How is the code's address determined for the call once it is available on the other machine?
  3. How are the arguments made available on the other machine?
  4. How are pointer arguments dealt with?
     - disallow the use of pointers
     - marshal/de-marshal the entire linked data structure (implies traversing all pointers) and copy the list
     - use *long pointers* between address spaces and fetch data only when the long pointer is dereferenced
  5. How are different data representations dealt with? E.g., big/little endian addressing, floating point representation, 7,8,16-bit characters.
  6. How are different languages dealt with?
  7. How are arguments and parameters type checked?
     - dynamic checks on each call $\Rightarrow$ additional runtime errors, uses additional CPU time to check and storage for type information.
     - have persistent symbol table information that describes the called interface, and can be located and used during compilation of the call.

- Some network-standards exist that convert data as it is transferred among computers.

- Some interface definition languages (IDL) exist that provide language-independent persistent interface definitions.

- Some systems provide a name-service to look up remote routines (begs the question of how do you find the name-service).

12.2 BSD Socket

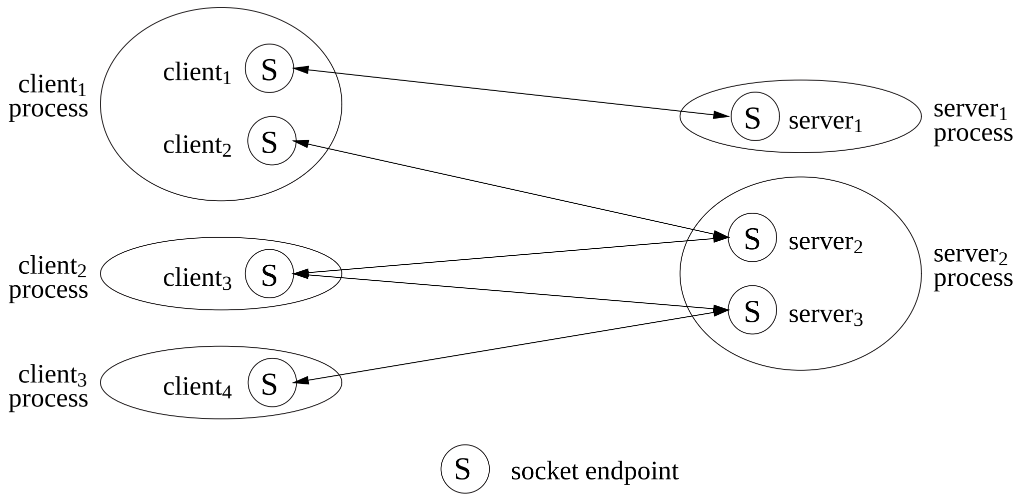
- The main mechanism for programs to push/pull information through a network is the socket.

- A **socket** is an end point for communicating among tasks in different processes, possibly on different computers.

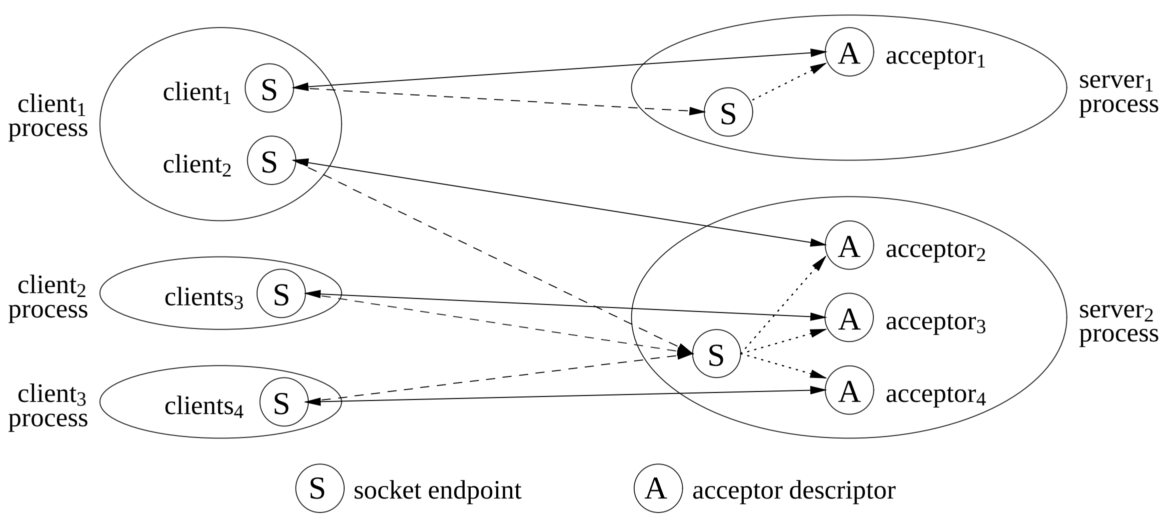
- An endpoint is accessed in one of two ways:

  1. **client**
     - one-to-many for connectionless communication with multiple server socket-endpoints
     - one to one for peer-connection communication with a server's acceptor socket-endpoint.

2. **server**
   - one-to-many for connectionless communication with multiple client socket-endpoints
   - one to one for peer-connection communication with a client's socket-endpoint.



- For connectionless communication, client endpoints can communicate bidirectionally with server endpoints, as long as the other's address is known.
- Possible because messages contain the address of the sender or receiver.
- Network routes the message to this address.
- For convenience, when a message arrives at a receiver, the sender's address replaces the receiver's address, so the receiver can reply back.

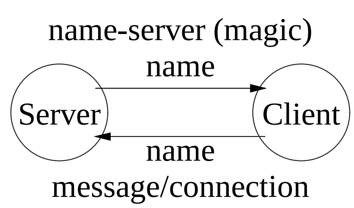


- For peer-connection communication, client endpoint can *only* communicate bidirectionally with server endpoint it has connected to.

- Dashed lines show the connection of the client and server.
- Dotted lines show the creation of an acceptor (worker) to service the connection for peer communication.
- Solid lines show the bidirectional communication among the client and server's acceptor.
- Messages do not contain sender and receive addresses, as these addresses are implicitly known through connection.
- Notice fewer socket endpoints in the peer-connection communication versus the connectionless communication, but more acceptors.
- For connectionless communication, a single socket-endpoint sequentially handles both the connection and the transfer of data for each message.
- For peer-connection communication, a single socket-endpoint handles connections and an acceptor transfers data in parallel.
- In general, peer-connection communication is more expensive (unless large amounts of data are transferred) but more reliable than connectionless communication.

12.2.1 Socket Names

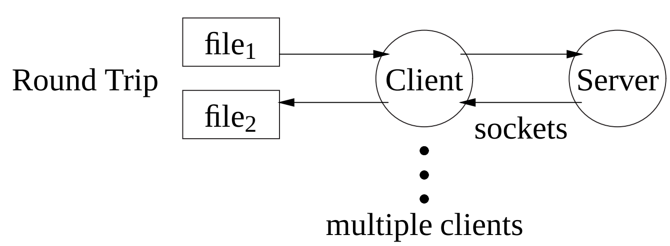
- Server socket has a name: character string for UNIX pipes or port-number/machine-address for an INET address.
- Clients must know this name to communicate.



- For connectionless communication, server receives messages containing the client's address.
- For peer-connection communication, server receives connection binding the client's address.

12.2.2 Round Trip

- Round trip example for connectionless and peer-connection communication:



- **μC++ hides many of the socket details and provides non-blocking I/O wherever possible.**

12.2.2.1 Peer-Connection (INET)

```
// CLIENT

const char EOD = '\377';
_Task Reader {
    uSocketClient &client;
    void main() {
        char buf[100];
        int len;
        for ( ;; ) { // EOD piggy-backed at end of last message
            len = client.read( buf, sizeof(buf) );
          if ( buf[len - 1] == EOD ) break;
            cout.write( buf, len );
        }
        cout.write( buf, len - 1 ); // do not write EOD
    }
  public:
    Reader( uSocketClient &client ) : client ( client ) {}
};

_Task Writer {
    uSocketClient &client;
    void main() {
        char buf[100];
        for ( ;; ) {
            cin.get( buf, sizeof(buf), '\0' );
          if ( buf[0] == '\0' ) break;
            client.write( buf, strlen( buf ) );
        }
        client.write( &EOD, sizeof(EOD) );
    }
  public:
    Writer( uSocketClient &client ) : client( client ) {}
};

void uMain::main() {
    uSocketClient client( atoi( argv[1] ), SOCK_STREAM ); // server connection
    Reader rd( client );
    Writer wr( client );
} // wait for rd/wr
```

```
// ACCEPTOR

_Task Acceptor {
    uSocketServer &sockserver;
    Server &server;
    void main() {
        try {
            uDuration timeout( 20, 0 ); // accept timeout
            uSocketAccept acceptor( sockserver, &timeout );// accept client connection
            char buf[100];
            int len;
            server.connection();        // tell server connected
            for ( ;; ) {
                len = acceptor.read( buf, sizeof(buf) );
                acceptor.write( buf, len );
              if ( buf[len - 1] == EOD ) break;
            } // for
            server.complete( this, false );  // terminate without timeout
        } catch( uSocketAccept::OpenTimeout ) {
            server.complete( this, true );   // terminate with timeout
        }
    }
  public:
    Acceptor( uSocketServer &socks, Server &server ) : sockserver( socks ), server( server ) {}
};
```

```
// SERVER

_Task Server {
    uSocketServer &sockserver;
    Acceptor *terminate;
    int acceptorCnt;
    bool timeout;
  public:
    Server( uSocketServer &socks ) : sockserver( socks ), acceptorCnt( 1 ), timeout( false ) {}
    void connection() {}
    void complete( Acceptor *terminate, bool timeout ) {
        Server::terminate = terminate; Server::timeout = timeout;
    }
  private:
```

```
        void main() {
            new Acceptor( sockserver, *this );    // create initial acceptor
            for ( ;; ) {
                _Accept( connection ) {
                    new Acceptor( sockserver, *this ); // create new acceptor after connection
                    acceptorCnt += 1;
                } or _Accept( complete ) {        // acceptor finished with client
                    delete terminate;             // delete acceptor
                    acceptorCnt -= 1;
              if ( acceptorCnt == 0 ) break;      // no outstanding connections, stop
                    if ( timeout ) {
                        new Acceptor( sockserver, *this ); // create new acceptor after a timeout
                        acceptorCnt += 1;
                    } // if
                } // _Accept
            } // for
        }
    };
    void uMain::main() {
        short unsigned int port;
        uSocketServer sockserver( &port, SOCK_STREAM ); // server connection to free port
        cout << port << endl; // print out free port for clients
        Server s( sockserver );   // execute until acceptor times out
    }
```

- Server keeps one acceptor pending, i.e., waiting for a connection.
- When a client connects with the pending acceptor, the server creates another one.
- If the pending acceptor times out $\Rightarrow$ no connections for N seconds so close server, unless still active connections.

12.3 Threads & Message Passing

- Message passing is an alternative mechanism to parameter passing.
- In message passing, information transmitted is (usually) 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, it might be possible to pass pointers.
- Message passing is usually direct communication.

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

task$_1$	**task**$_2$
send(tid$_2$, msg)	receive(tid$_1$, msg)

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

task$_1$	**task**$_2$
send(tid$_2$, msg)	tid = receive(msg)

12.3.1 Nonblocking Send

- Send does not block, and receive gets the messages in the order they are sent (SendNonBlock).
  - $\Rightarrow$ 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() {                         Consumer() {
    . . .                                . . .
    for (;;) {                           for (;;) {
        produce an item                      Receive(PId,msg);
        SendNonBlock(CId,msg);               consume the item
    }                                    }
}                                    }
```

12.3.2 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

12.3.3 Send-Receive-Reply

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

```
Producer() {                         Consumer() {
    . . .                                . . .
    for (;;) {                           for (;;) {
        produce an item                      prod = ReceiveReply(msg);
        SendReply(CId,ans,msg);              consume the item
        . . .                                Reply(prod,ans)  never
    }                                    }                    blocks
}                                    }
```

- 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;
}
```

12.3.4 Message Format

- variable-size messages
    - complex implementation, easy to use
- fixed-size message
    - 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***
- byte stream
    - no visible message boundaries at receiver
    - emulate standard I/O stream
    - efficient batching of small messages

12.3.5 Communication Exceptions

- Assume send-receive-reply.
- Sender dies ⇒ receiver blocks forever.
- Receiver dies ⇒ sender blocks forever.
- Defective communication:

lost message - detected by time-out waiting for reply by sender

damaged message - detected by error check at sender or receiver

out-of-order message - detected by receiver using information attached to the message

duplicate message - detected by receiver using information attached to the message

- In the first case, the sender retransmits the message.
- In the second case, the receiver could request retransmission.
- But if it just ignores the message, it looks like the first case to the sender.
- Reordering messages and discarding duplicates can be done entirely at the receiver.
- ***But:*** Sender can never distinguish between lost message and dead receiver.

12.4 MPI

- **Message Passing Interface** (MPI) is message-passing library-interface specification.
- MPI addresses the message-passing programming-model, where data is transfered among the address spaces of processes for computation.
- A message-passing specification provides portability across heterogeneous computer-clusters/networks or on shared-memory multiprocessor computers.
- MPI is intended to deliver high performance with low latency and high bandwidth without burdening users with details of the network.
- MPI provide point-to-point send/receive operations, exchanging data among processes.
- Data types: basic types, arrays and structure (no pointers).
- Data types must match between sender and receiver.

MPI datatype	C	MPI datatype	C
MPI_CHAR	**char**	MPI_UNSIGNED_LONG	**unsigned long**
MPI_WCHAR	**wchar_t**	MPI_UNSIGNED_LONG_LONG	**unsigned long long**
MPI_SHORT	**signed short**	MPI_FLOAT	**float**
MPI_INT	**signed int**	MPI_DOUBLE	**double**
MPI_LONG	**signed long**	MPI_LONG_DOUBLE	**long double**
MPI_LONG_LONG	**signed long long**	MPI_C_BOOL	_Bool
MPI_SIGNED_CHAR	**signed char**	MPI_C_COMPLEX	**float** _Complex
MPI_UNSIGNED_CHAR	**unsigned char**	MPI_C_DOUBLE_COMPLEX	**double** _Complex
MPI_UNSIGNED_SHORT	**unsigned short**	MPI_BYTE	
MPI_UNSIGNED	**unsigned int**	MPI_PACKED	

- Point-to-point operations can be synchronous, asynchronous or buffered.

- MPI Send and Recv are the two most used constructs (C++ binding is deprecated):

```
int MPI_Send(
    const void *buf,               // send buffer
    int count,                     // # of elements to send
    MPI_Datatype datatype,         // type of elements
    int dest,                      // destination rank
    int tag,                       // message tag
    MPI_Comm communicator          // normally MPI_COMM_WORLD
);
int MPI_Recv(
    void *buf,                     // receive buffer
    int count,                     // # of elements to receive
    MPI_Datatype datatype,         // type of elements
    int source,                    // source rank
    int tag,                       // message tag
    MPI_Comm communicator,         // normally MPI_COMM_WORLD
    MPI_Status *status             // status object or NULL
);

#include <mpi.h>
enum { PROD = 0, CONS = 1 };
void producer() {
    int buf[10];                          // transmit array to consumer
    for ( int i = 0; i < 20; i += 1 ) {   // generate N arrays
        for ( int j = 0; j < 10; j += 1 ) buf[j] = i;
        MPI_Send( buf, 10, MPI_INT, CONS, 0, MPI_COMM_WORLD );
    }
    buf[0] = -1;                          // send sentinel value
    MPI_Send( buf, 1, MPI_INT, CONS, 0, MPI_COMM_WORLD ); // only send 1
}

void consumer() {
    int buf[10];
    for ( ;; ) {
        MPI_Recv( buf, 10, MPI_INT, PROD, 0, MPI_COMM_WORLD, NULL );
      if ( buf[0] == -1 ) break;          // receive sentinel value ?
        for ( int j = 0; j < 10; j += 1 ) printf( "%d", buf[j] );
        printf( "\n" );
    }
}

int main( int argc, char *argv[] ) {
    MPI_Init( &argc, &argv );             // initializing MPI
    int rank;
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if ( rank == PROD ) producer(); else consumer();
    MPI_Finalize();
}
```

- Two copies are started: one for producer and one for consumer, distinguished by *rank*.

- Compile and run using MPI environment:

```
$ mpicc bb.cc              # use C compiler
$ mpirun -np 2 a.out       # 2 processes needed (must match)
```

- Blocking operations:
  - send: send specific (CONS), message specific (0), block until buffer free

    ```
    MPI_Send( &v, 1, MPI_INT, CONS, 0, MPI_COMM_WORLD );
    ```
  - receive: receive specific (PROD), message specific (0)

    ```
    MPI_Recv( &v, 1, MPI_INT, PROD, 0, MPI_COMM_WORLD, NULL );
    ```
  - receive: receive any, message specific (0)

    ```
    MPI_Recv( &v, 1, MPI_INT, MPI_ANY_SOURCE, 0,
                  MPI_COMM_WORLD, NULL );
    ```
  - receive: receive specific, message any

    ```
    MPI_Recv( &v, 1, MPI_INT, PROD, MPI_ANY_TAG,
                  MPI_COMM_WORLD, NULL );
    ```
  - receive: receive any, message any

    ```
    MPI_Recv( &v, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                  MPI_COMM_WORLD, NULL );
    ```
- Emulate send/receive/reply with send/receive (double interface cost)

```
enum { PROD = 0, CONS = 1 };
void producer() {
    int v, a;
    for ( v = 0; v < 5; v += 1 ) {          // generate values
        MPI_Send( &v, 1, MPI_INT, CONS, 0, MPI_COMM_WORLD ); // send
        MPI_Recv( &a, 1, MPI_INT, CONS, 0, MPI_COMM_WORLD, NULL ); // block for reply
    }
    v = -1;                                 // send sentinel value
    MPI_Send( &v, 1, MPI_INT, CONS, 0, MPI_COMM_WORLD );
    MPI_Recv( &a, 0, MPI_INT, CONS, 0, MPI_COMM_WORLD, NULL );
}
void consumer() {
    int v, a;
    for ( ;; ) {
        MPI_Recv( &v, 1, MPI_INT, PROD, 0, MPI_COMM_WORLD, NULL );
        // examine message
      if ( v == -1 ) break;                 // receive sentinel value ?
        a = v + 100;
        MPI_Send( &a, 1, MPI_INT, PROD, 0, MPI_COMM_WORLD );
    }
    MPI_Send( &a, 0, MPI_INT, PROD, 0, MPI_COMM_WORLD );
}
```

- Nonblocking operations (future like):

- send: send specific (CONS), message specific (0)

```
MPI_Request req;
MPI_Isend( buf, 10, MPI_INT, CONS, 0, MPI_COMM_WORLD, &req );
// work during send
MPI_Wait( &req, NULL ); // wait for send to complete
```

- receive: receive specific (PROD), message specific (0)

```
MPI_Irecv( buf, 10, MPI_INT, PROD, 0, MPI_COMM_WORLD, &req );
// work during receive
MPI_Wait( &req, NULL ); // wait for receive to complete
```

- wait for *any* request in an array of requests

```
MPI_Request reqs[20];  // outstanding requests
int index;
MPI_Waitany( 20, reqs, &index, NULL );  // wait for any receive
```

- wait for *all* request in an array of requests

```
int buf[20][10];            // all data
MPI_Request reqs[20];  // outstanding requests
MPI_Waitall( 20, reqs, NULL ); // wait for all receives
```

```
void producer() {
    int buf[10];
    MPI_Request r;
    for ( int i = 0; i < 20; i += 1 ) {      // generate N arrays
        for ( int j = 0; j < 10; j += 1 ) buf[j] = i;
        MPI_Isend( buf, 10, MPI_INT, CONS, 0, MPI_COMM_WORLD, &r );
        // work during send
        MPI_Wait( &r, NULL );             // wait for send to complete
    }
    buf[0] = -1;
    MPI_Isend( buf, 1, MPI_INT, CONS, 0, MPI_COMM_WORLD, &r ); // only send 1
    MPI_Wait( &r, NULL );
}
void consumer() {
    int buf[10];
    MPI_Request r;
    for ( ;; ) {
        MPI_Irecv( buf, 10, MPI_INT, PROD, 0, MPI_COMM_WORLD, &r );
        // work during receive
        MPI_Wait( &r, NULL );              // wait for receive to complete
      if ( buf[0] == -1 ) break;            // sentinel value ?
        for ( unsigned int j = 0; j < 10; j += 1 ) printf( "%d", buf[j] );
        printf( "\n" );
    }
}
```

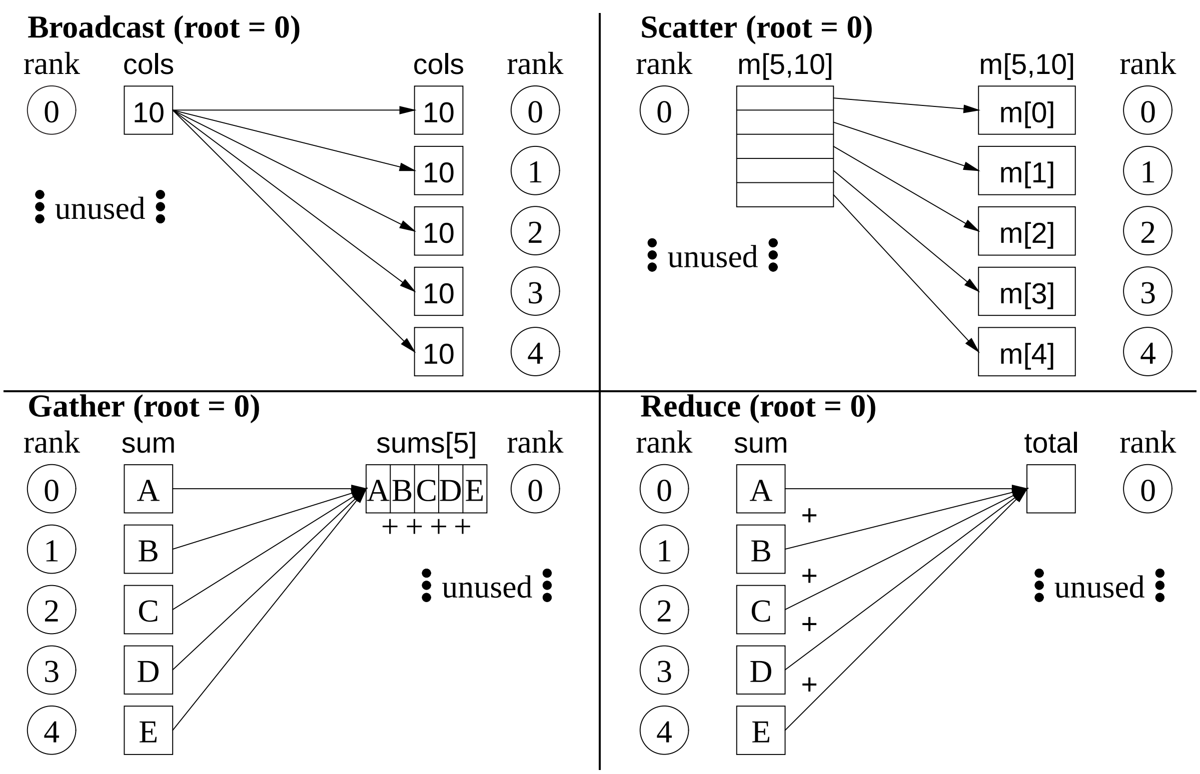
- Bcast, Scatter, Gather, Reduce : powerful synchronization/communication via rendezvous.

```
enum { root = 0 };
int main( int argc, char *argv[] ) {
    int rows = 5, cols = 10, sum = 0, total = 0;
    int m[rows][cols], sums[rows];
    MPI_Init( &argc, &argv );
    int rank;
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if ( rank == root ) { /* read / generate matrix */ }
    MPI_Bcast( &cols, 1, MPI_INT, root, MPI_COMM_WORLD );
    MPI_Scatter( m, cols, MPI_INT, m[rank], cols, MPI_INT, root, MPI_COMM_WORLD );
    for ( int i = 0; i < cols; i += 1 ) sum += m[rank][i]; // sum specific row
    MPI_Gather( &sum, 1, MPI_INT, sums, 1, MPI_INT, root, MPI_COMM_WORLD );
    if ( rank == root ) {                    // compute total
        for ( int i = 0; i < rows; i += 1 ) total += sums[i];
        printf( "total:%d\n", total );
    }
    MPI_Reduce( &sum, &total, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD );
    if ( rank == root ) { printf( "total:%d\n", total ); }
    MPI_Finalize();
}
```

	matrix				subtotals
$T_0\sum$	23	10	5	7	0
$T_1\sum$	-1	6	11	20	0
$T_2\sum$	56	-13	6	0	0
$T_3\sum$	-2	8	-5	1	0
				total	$\sum$

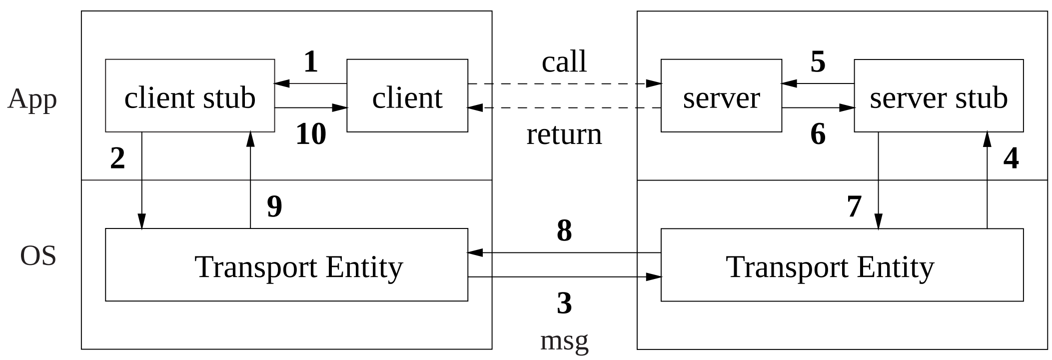
- Bcast, Scatter, Gather, Reduce treat root process as sender/receiver, and other processes as receiver/sender.

- E.g., for Bcast, root writes from cols and other processes read into cols.



12.5 Remote Procedure Call (RPC)

- RPC transparently allows a call from client to server in another address space (or another machine), with possible return value, in a type-safe way.
- RPC works across different hardware platforms and operating systems.
- Standard data representation: *External Data Representation* (XDR).
- Alternative representation: Java/C++ XML-RPC interfaces.
- Blueprint for Java RMI, SOAP, CORBA, etc.



1 local call from client to client stub

2 client stub collects arguments into a message (marshalling) and passes it to the local transport (may be an OS call)

3 client transport entity sends message to server transport entity

4 server transport entity passes message to server stub

5 server stub unmarshalls arguments and calls local routine

6 server performs its processing and returns to server stub

7-10 marshall return value(s) into message, pass back to client in similar fashion.

12.5.1 RPCGEN

- RPCGEN is an interface generator pre-compiler to simplify RPC.
- **Interface-definition file** to create client and server stubs in C (file calculator.x).

```
program CALCULATOR {                    /* create remote calculator */
    version INTERFACE {                 /* calculator interface */
        int add( int, int ) = 1;        /* number interface routines */
        int sub( int, int ) = 2;
        int mul( int, int ) = 3;
        int div( int, int ) = 4;
    } = 1;                              /* version number */
} = 0x20000000;                         /* program number (unique) */
```

- ◦ **char** * written as string
- ◦ arguments passed by value, **but return value becomes pointer**
- ◦ interface routines are numbered (you decide on numbering)
- ◦ version number incremented when functionality changes (multiple versions possible)
- ◦ program number uniquely defines program for authenticating client/server calls

- Server defines routines (file server.cc):

```
#include "calculator.h"   // remote interfaces from rpcgen

int *add_1_svc( int arg1, int arg2, struct svc_req *rqstp ) {
    static int result;   result = arg1 + arg2;   return &result;
}
int *sub_1_svc( int arg1, int arg2, struct svc_req *rqstp ) {
    static int result;   result = arg1 - arg2;   return &result;
}
int *mul_1_svc( int arg1, int arg2, struct svc_req *rqstp ) {
    static int result;   result = arg1 * arg2;   return &result;
}
int *div_1_svc( int arg1, int arg2, struct svc_req *rqstp ) {
    static int result;   result = arg1 / arg2;   return &result;
}
```

- ◦ Return type must be a pointer to storage that outlives the routine call (use **static** trick).
- ◦ Suffix "_N_svc" is version number appended to routine name.
- ◦ Extra parameter rqstp has information about invocation, e.g., program, version, routine numbers, etc.

- Client makes remote calls to server (file client.cc):

```
#include "calculator.h"   // remote interfaces from rpcgen
#include <iostream>
using namespace std;

int main( int argc, const char *argv[] ) {
    if ( argc != 2 ) {
        cerr << "usage: " << argv[0] << "server-hostname" << endl;
        exit( EXIT_FAILURE );
    }
    CLIENT *clnt = clnt_create( argv[1],
                                CALCULATOR, INTERFACE, "udp" );
    if ( clnt == NULL ) {
        clnt_pcreateerror( argv[1] );
        exit( EXIT_FAILURE );
    }
```

```
    for ( ;; ) {
        int a, b, *result;
        char op;
        cin >> a >> op >> b;
      if ( cin.fail() ) break;
        switch (op) {
          case '+': result = add_1( a, b, clnt ); break; // remote calls
          case '-': result = sub_1( a, b, clnt ); break;
          case '*': result = mul_1( a, b, clnt ); break;
          case '/': result = div_1( a, b, clnt ); break;
          default: cerr << op << " unsupported" << endl; continue;
        }
        if ( result == NULL ) clnt_perror( clnt, "call failed" );
        else cout << "result: " << *result << endl;
    }
    clnt_destroy( clnt );
}
```

- clnt_create connects to server using UDP socket and returns client handle.
- client handle is passed into the stub routine (without "_svc") as last argument, which calls remote routine in server
- client handle allows server to get back to client
- pointer to the result is returned and must be checked for failure
- destroy client handle to close connection to server

- Compiling interface-definition/client/server:

```
$ rpcgen -N calculator.x      # .x suffix
```

  - -N allows multiple parameters and pass by value
  - generate server stub-routines and communication code in file calculator_svc.c to accept remote client calls and return results
  - generate client stub-routines in file calculator_clnt.c to transfer arguments/return-value to/from server
  - generate RPC message-protocol in file calculator_xdr.c for remote communication
  - generate all stub interfaces in file calculator.h for inclusion in client and server

```
$ g++ client.cc calculator_clnt.c calculator_xdr.c -o client -lnsl
$ g++ server.cc calculator_svc.c calculator_xdr.c -o server -lnsl
```

- OS must be running daemon portmap to facilitate rendezvous between server and client.

- Usage of client and server on two different computers:

computer1	computer2
$ server &	$ client computer1.locale.ca
[1] 23895	2+5
. . .	result: 7
$ kill 23895	34-77
	result: -43
	42*42
	result: 1764
	7/3
	result: 2
	C-d

Index