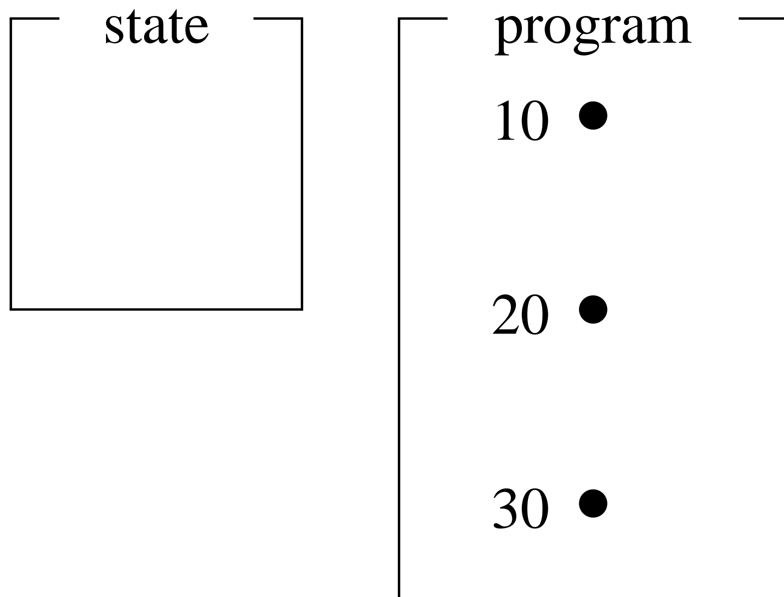
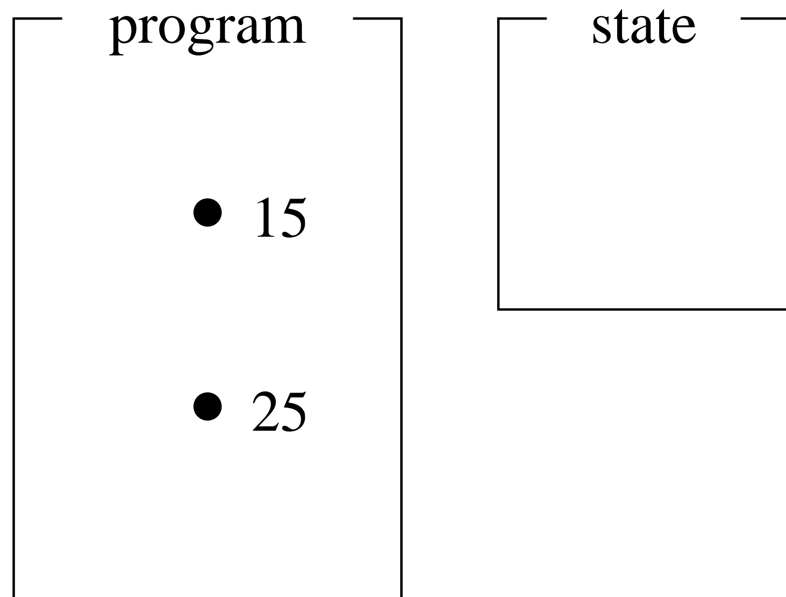
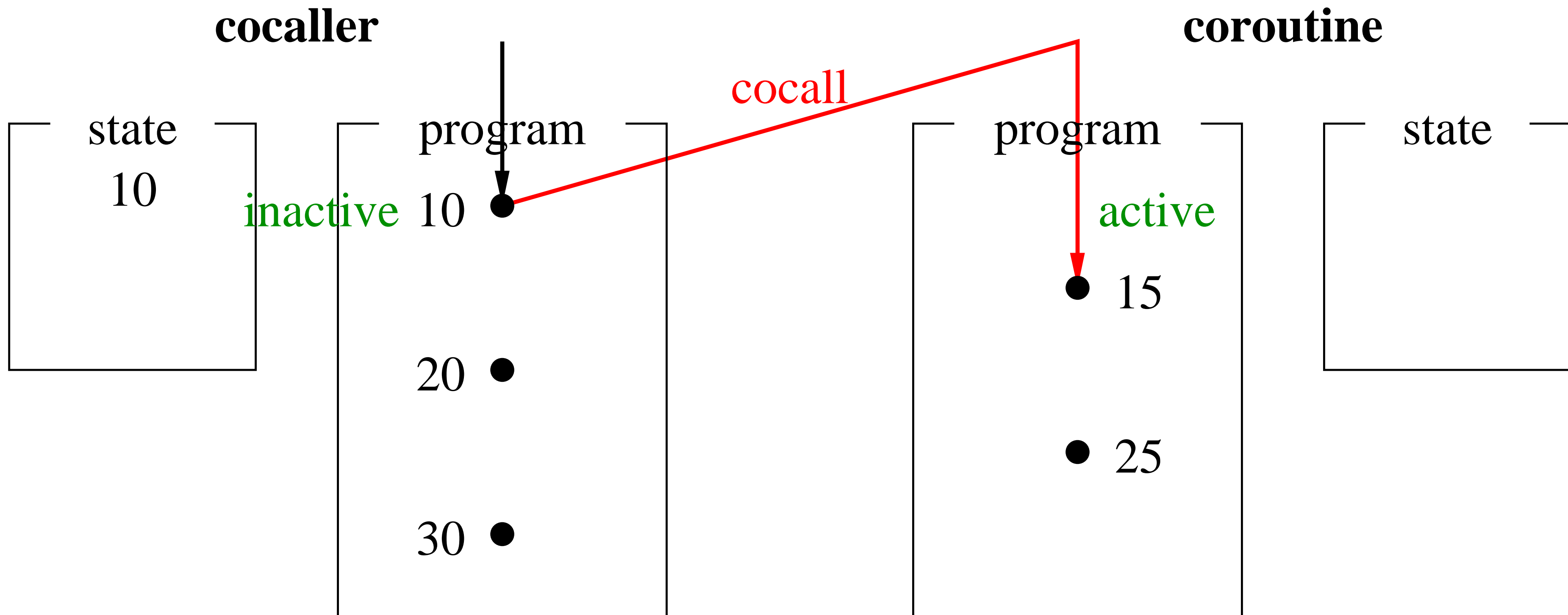
2 Coroutine

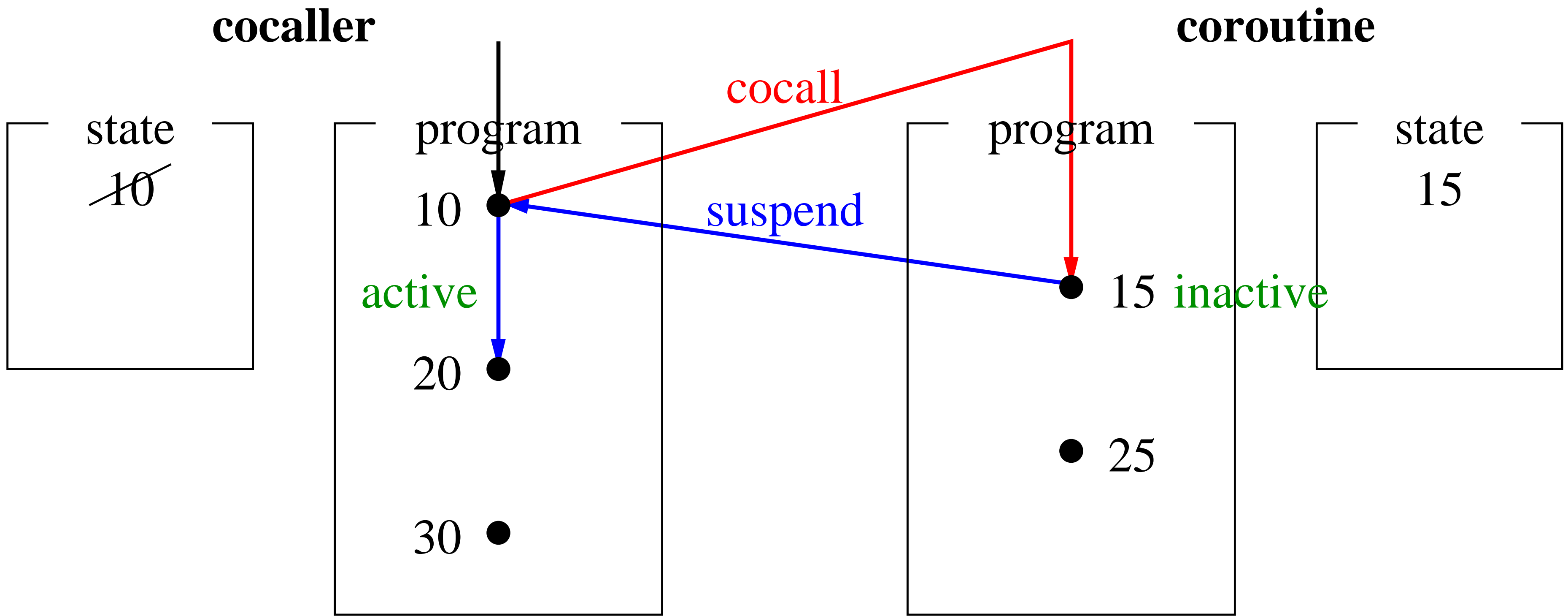
- A **coroutine** is a routine that can also be suspended at some point and resume from that point when control returns.
- The state of a coroutine consists of:
  - an **execution location**, starting at the beginning of the coroutine and remembered at each suspend.
  - an **execution state** holding the data created by the code the coroutine is executing.
    $\Rightarrow$ 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.

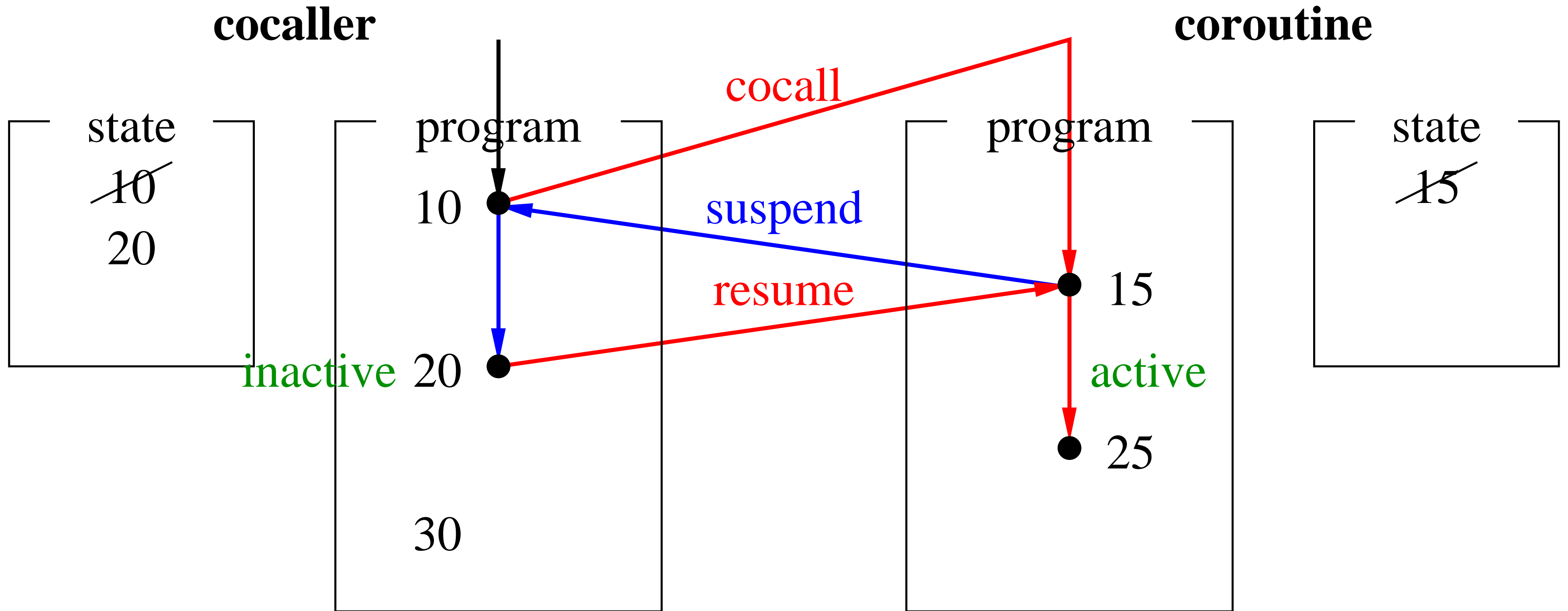
Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 License.

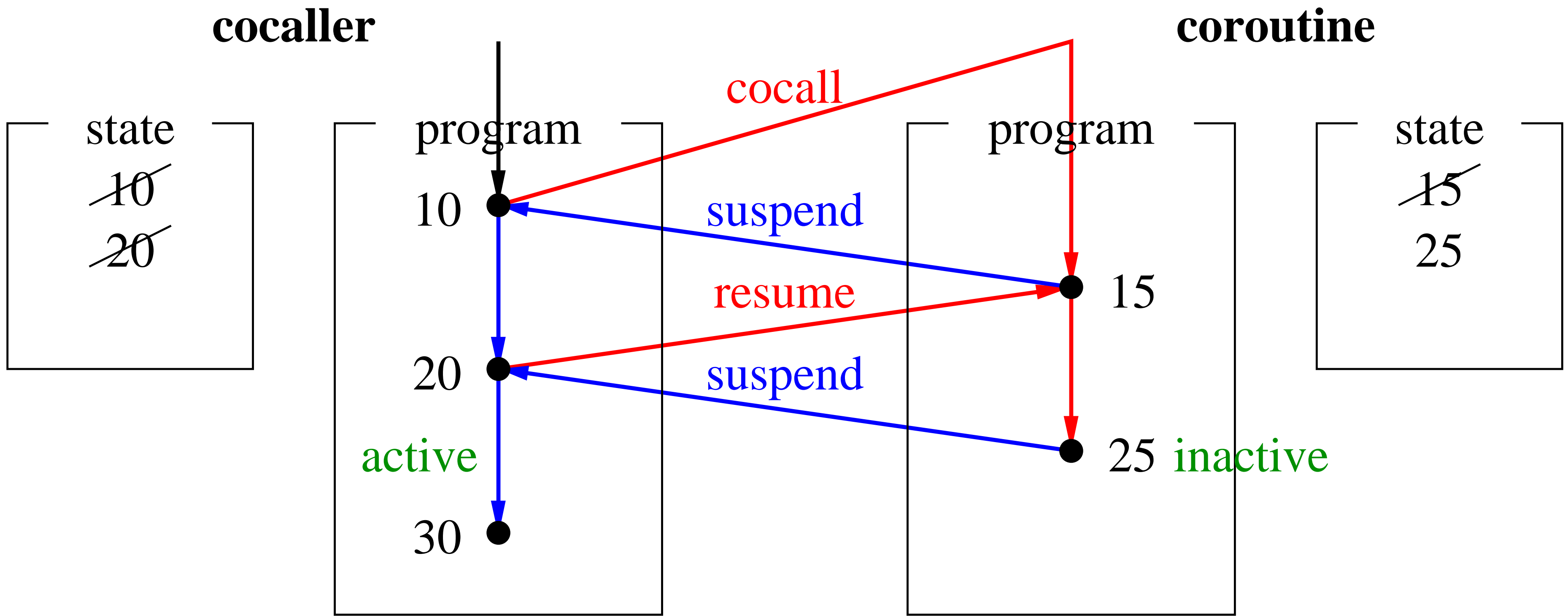

cocaller
state
program
10
20
30

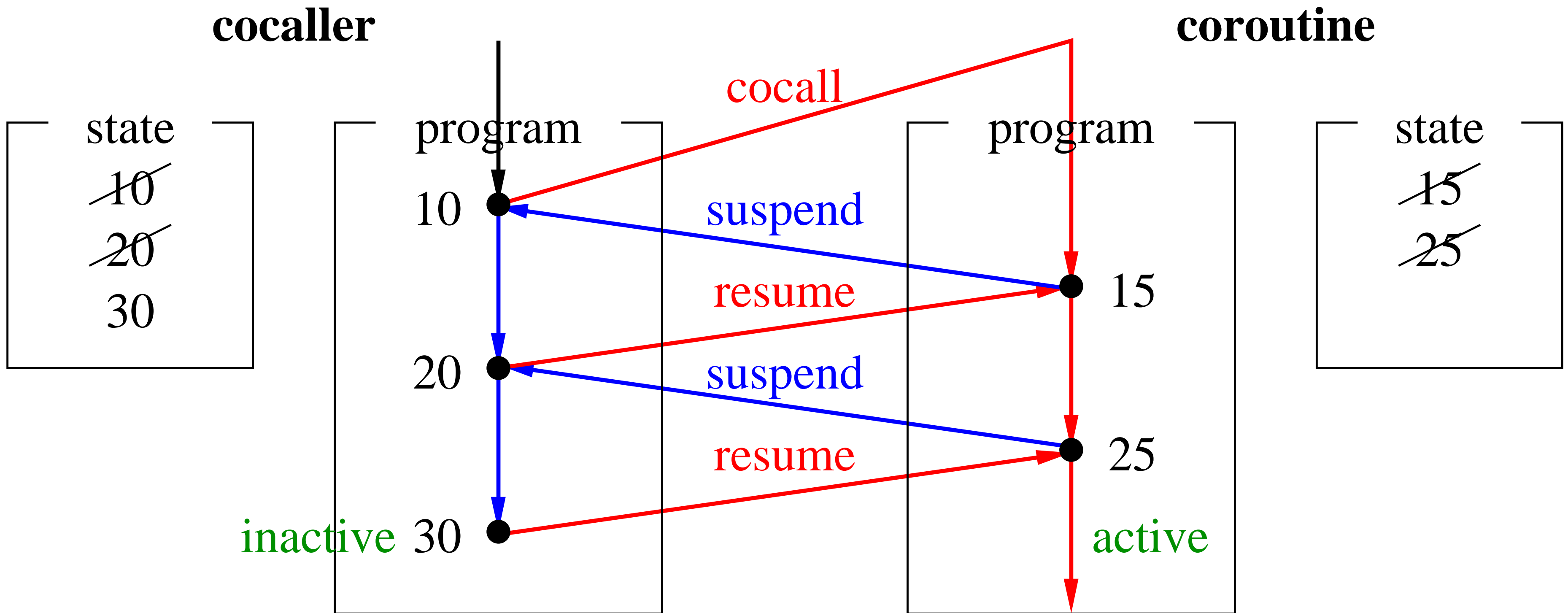

coroutine
program
state
15
25

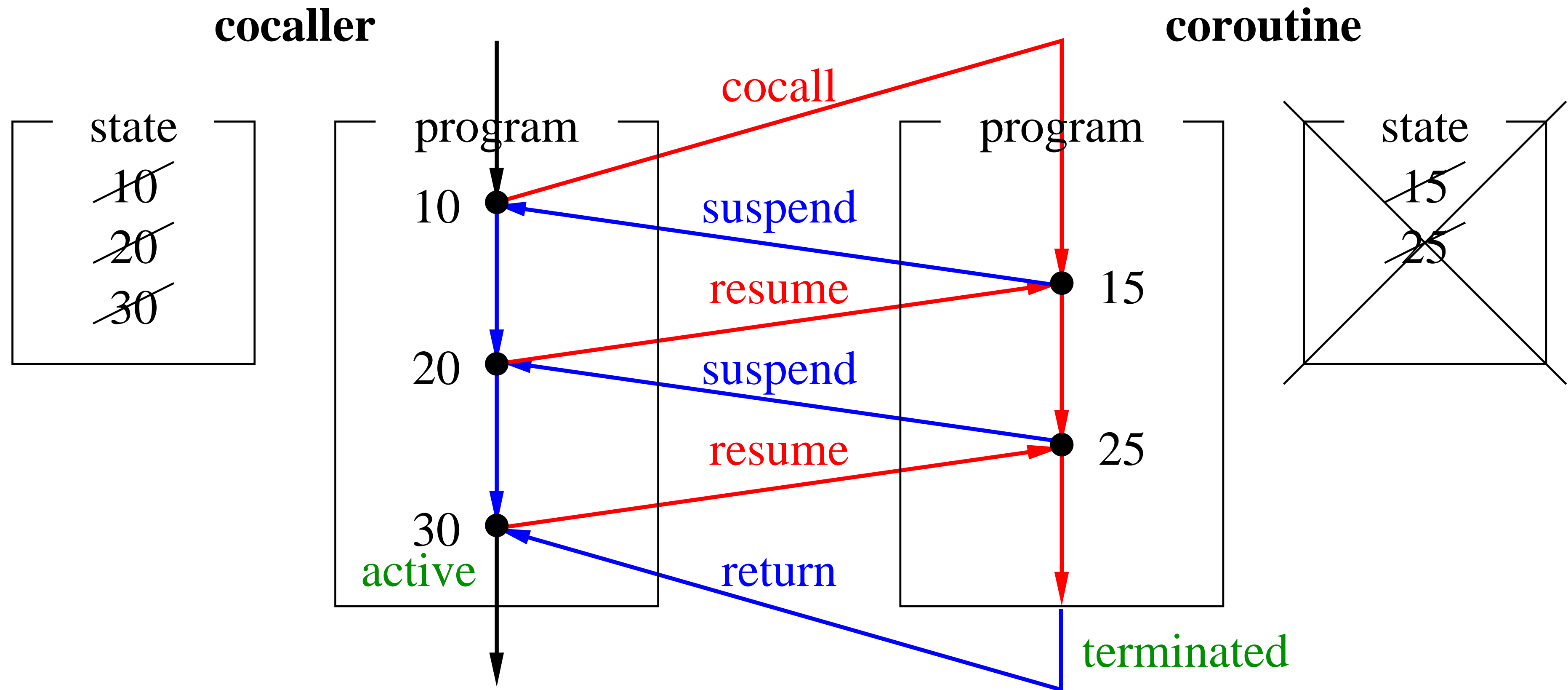

cocaller
coroutine
cocall
state
10
program
inactive
10
20
30
program
active
15
25
state


cocaller
coroutine
cocall
state
10
program
10
20
30
active
suspend
program
15
25
inactive
state
15


cocaller
coroutine
cocall
state
10
20
program
10
20
30
suspend
resume
inactive
program
15
active
25
state
15


cocaller
coroutine
cocall
state
10
20
program
10
suspend
resume
20
suspend
active
30
program
15
25
inactive
state
15
25


cocaller
coroutine
cocall
state
program
program
state
10
20
30
15
25
10
suspend
resume
suspend
resume
15
20
25
inactive 30
active



- A coroutine handles the class of problems that need to retain state between calls (e.g. plugin, device driver, finite-state machine).
- A coroutine executes synchronously with other coroutines; hence, no concurrency among coroutines.
- Coroutines are the precursor to concurrent tasks, and introduce the complex concept of suspending and resuming on separate stacks.

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

2.1 Semi-Coroutine

2.1.1 Fibonacci Sequence

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

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

2.1.1.1 Direct

- Assume output can be generated at any time.

```
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.

2.1.1.2 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

2.1.1.3 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
}
```

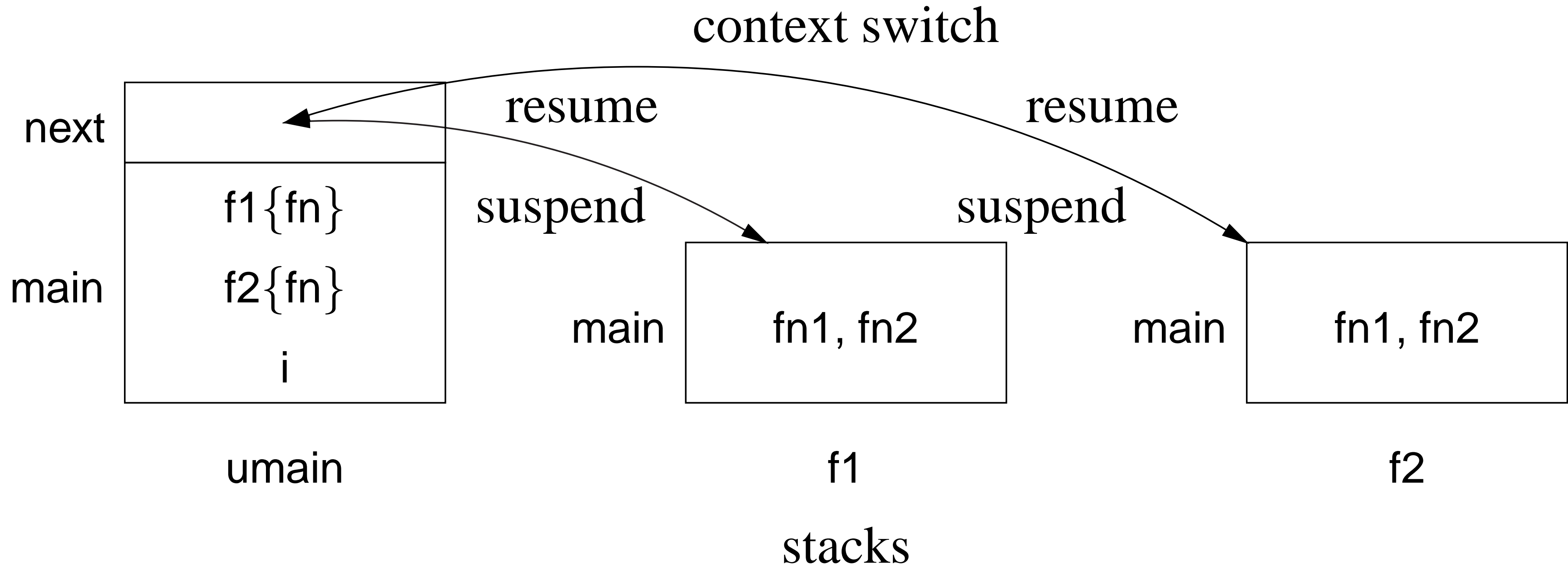
- unencapsulated program global variables becomes encapsulated object global variables
- multiple fibonacci generators (objects) can run at a time
- execution state must still be explicitly retained

2.1.1.4 Coroutine

```
#include <uC++.h>           // first include file
#include <iostream>
using namespace std;
_Coroutine fibonacci {      // : public uBaseCoroutine
    int fn;                 // used for communication
    void main() {           // distinguished member
        int fn1, fn2;       // retained between resumes
        fn = 0;  fn1 = fn;
        suspend();          // return to last resume
        fn = 1;  fn2 = fn1;  fn1 = fn;
        suspend();          // return to last resume
        for ( ;; ) {
            fn = fn1 + fn2;  fn2 = fn1;  fn1 = fn;
            suspend();      // return to last resume
        }
    }
  public:
    int next() {
        resume();           // transfer to last suspend
        return fn;
    }
};
```

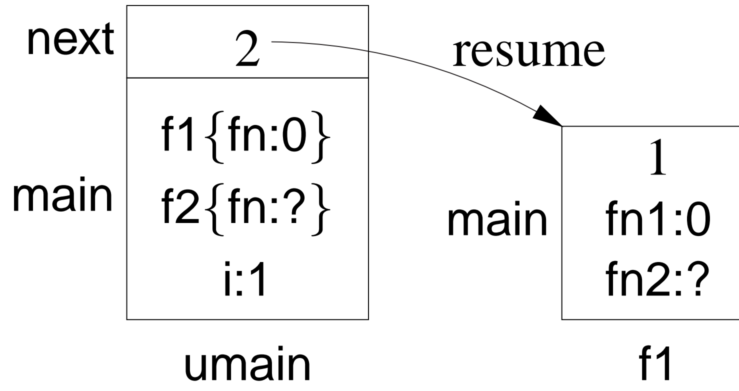
```
void uMain::main() {          // argc, argv class variables
    fibonacci f1, f2;
    for ( int i = 1; i <= 10; i += 1 ) {
        cout << f1.next() << " " << f2.next() << endl;
    }
}
```

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

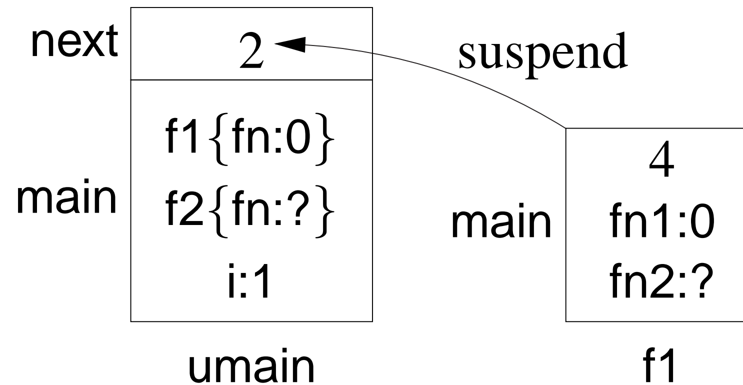

context switch
resume
resume
suspend
suspend
next
main
f1{fn}
f2{fn}
i
main
fn1, fn2
main
fn1, fn2
umain
f1
f2
stacks

```
_Coroutine fibonacci {
    int fn;
    void main() {
        int fn1, fn2;
        fn = 0; fn1 = fn;
        suspend();
        fn = 1; fn2 = fn1; fn1 = fn;
        suspend();
        for ( ;; ) {
            fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
            suspend();
        }
    }
  public:
    int next() {
        resume();
        return fn;
    }
};
```

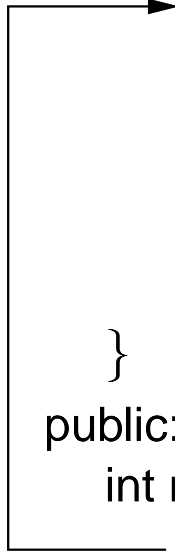
1st

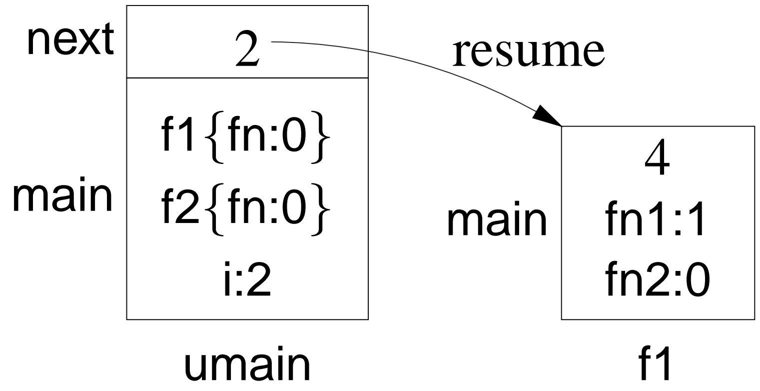


```
_Coroutine fibonacci {
    int fn;
    void main() {
        int fn1, fn2;
        fn = 0; fn1 = fn;
        suspend();
        fn = 1; fn2 = fn1; fn1 = fn;
        suspend();
        for ( ;; ) {
            fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
            suspend();
        }
    }
  public:
    int next() {
        resume();
        return fn;
    }
};
```

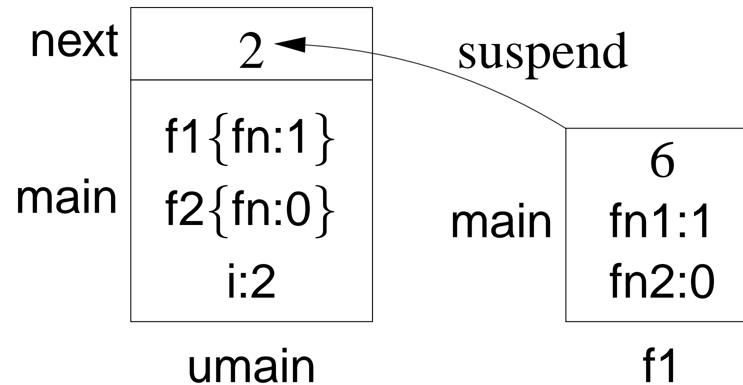


```
_Coroutine fibonacci {
    int fn;
    void main() {
        int fn1, fn2;
        fn = 0; fn1 = fn;
        suspend();
        fn = 1; fn2 = fn1; fn1 = fn;
        suspend();
        for ( ;; ) {
            fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
            suspend();
        }
    }
  public:
    int next() {
        resume();
        return fn;
    }
};
```



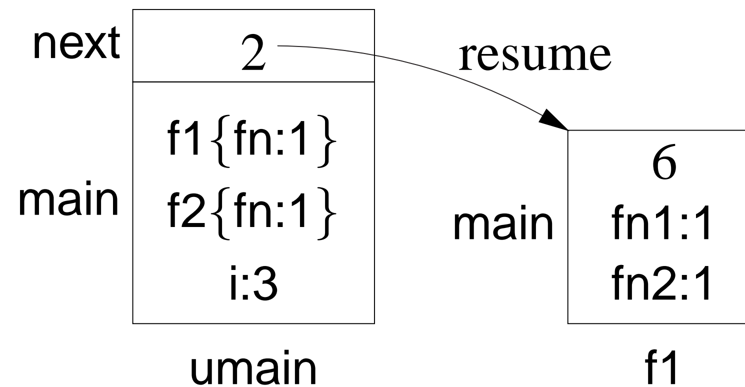


```
_Coroutine fibonacci {
    int fn;
    void main() {
        int fn1, fn2;
        fn = 0; fn1 = fn;
        suspend();
        fn = 1; fn2 = fn1; fn1 = fn;
        suspend();
        for ( ;; ) {
            fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
            suspend();
        }
    }
  public:
    int next() {
        resume();
        return fn;
    }
};
```

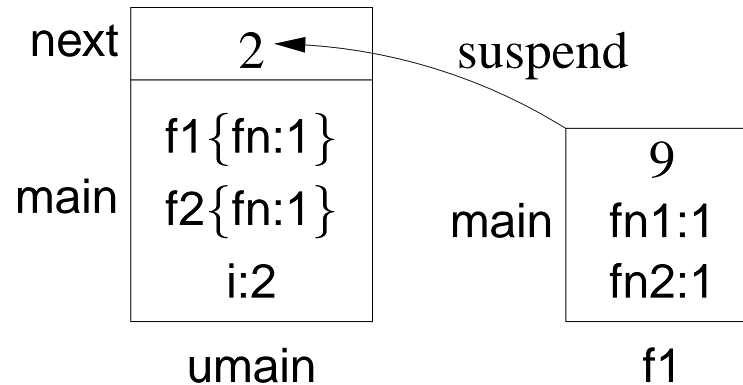


```
_Coroutine fibonacci {
    int fn;
    void main() {
        int fn1, fn2;
        fn = 0; fn1 = fn;
        suspend();
        fn = 1; fn2 = fn1; fn1 = fn;
        suspend();
        for ( ;; ) {
            fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
            suspend();
        }
    }
  public:
    int next() {
        resume();
        return fn;
    }
};
```

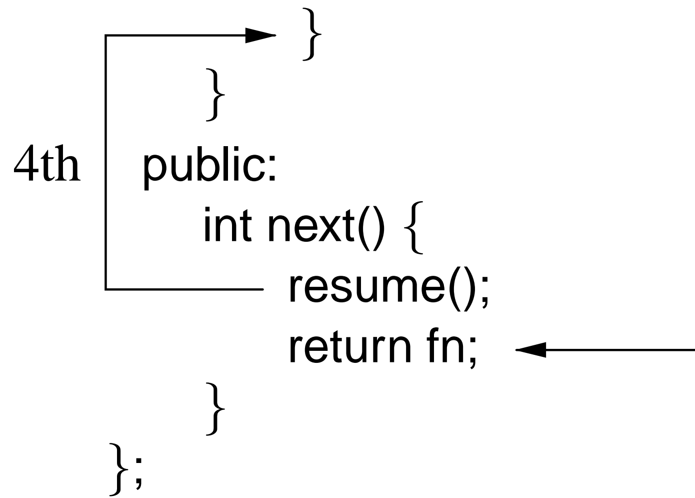
3rd

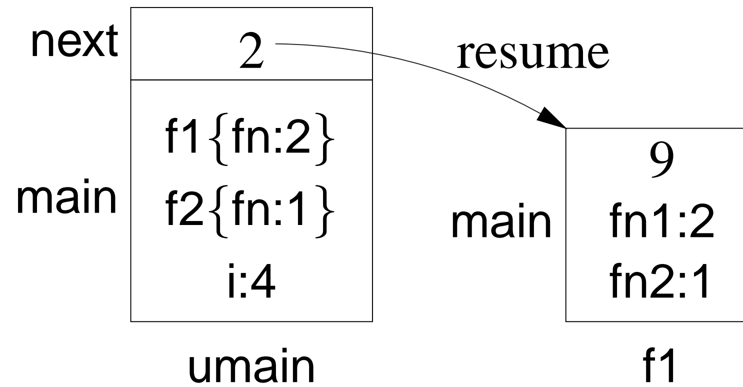


```
_Coroutine fibonacci {
    int fn;
    void main() {
        int fn1, fn2;
        fn = 0; fn1 = fn;
        suspend();
        fn = 1; fn2 = fn1; fn1 = fn;
        suspend();
        for ( ;; ) {
            fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
            suspend();
        }
    }
  public:
    int next() {
        resume();
        return fn;
    }
};
```



```
_Coroutine fibonacci {
    int fn;
    void main() {
        int fn1, fn2;
        fn = 0; fn1 = fn;
        suspend();
        fn = 1; fn2 = fn1; fn1 = fn;
        suspend();
        for ( ;; ) {
            fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
            suspend();
        }
    }
  public:
    int next() {
        resume();
        return fn;
    }
};
```





- routine frame at the top of the stack *knows* where to activate execution
- coroutine main does not have to return before coroutine object is deleted
- uMain is the initial coroutine started by μC++
- argc and argv are implicitly defined in uMain::main
- <uC++.h> must be first include
- compile with u++ command

2.1.2 Formatted Output

Unstructured input:

```
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.

2.1.2.1 Direct

- Assume input can be obtained at any time.

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

    for ( ;; ) {                        // for as many characters
        for ( g = 0; g < 5; g += 1 ) { // groups of 5 blocks
            for ( b = 0; b < 4; b += 1 ) { // blocks of 4 chars
                cin >> ch;      // read one character
      if ( cin.eof() ) goto fini;  // eof ? multi-level exit
                cout << ch;     // print character
            }
            cout << "   ";      // print block separator
        }
        cout << endl;               // print group separator
    }
  fini: ;
    if ( g != 0 || b != 0 ) cout << endl; // special case
}
```

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

2.1.2.2 Routine

```
int g, b;                          // global variables
void fmtLines( char ch ) {
    if ( ch == EOF ) { cout << endl; return; }
    if ( ch == '\n' ) return;      // ignore newline characters
    cout << ch;                    // print character
    b += 1;
    if ( b == 4 ) {                // block of 4 chars
        cout << "  ";              // block separator
        b = 0;
        g += 1;
    }
    if ( g == 5 ) {                // group of 5 blocks
        cout << endl;              // group separator
        g = 0;
    }
}
```

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

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

2.1.2.3 Class

```
class FmtLines {
        int g, b;                           // global class variables
    public:
        FmtLines() : g( 0 ), b( 0 ) {}
        ~FmtLines() { if ( g != 0 || b != 0 ) cout << endl; }
        void prt( char ch ) {
            cout << ch;                     // print character
            b += 1;
            if ( b == 4 ) {                 // block of 4 chars
                cout << "  ";               // block separator
                b = 0;
                g += 1;
            }
            if ( g == 5 ) {                 // group of 5 blocks
                cout << endl;               // group separator
                g = 0;
            }
        }
};
```

```
int main() {
    FmtLines fmt;
    char ch;
    for ( ;; ) {                    // for as many characters
        cin >> ch;                  // read one character
      if ( cin.eof() ) break;       // eof ?
        fmt.prt( ch );
    }
}
```

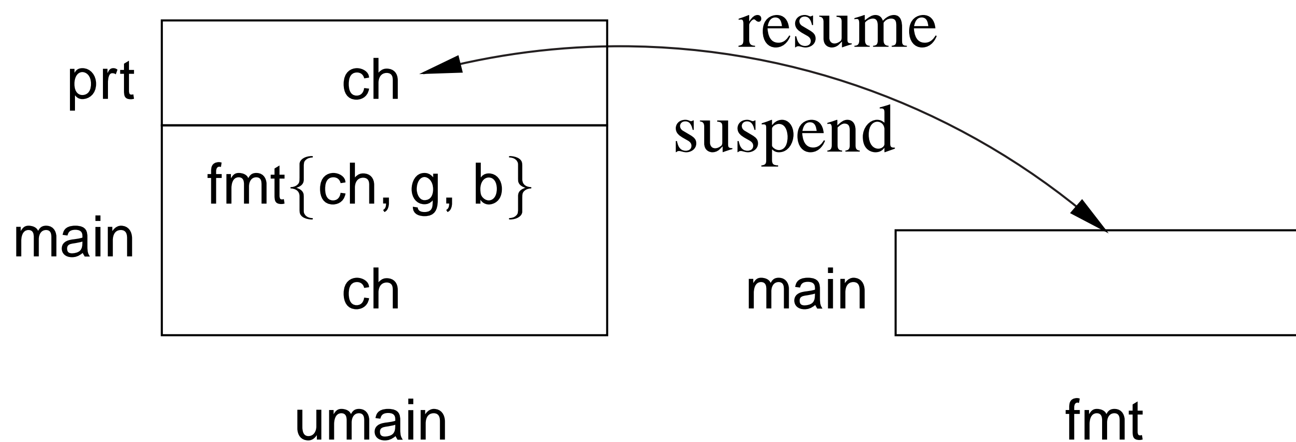
- Solves encapsulation and multiple instances issues, but still explicitly managing execution state.

2.1.2.4 Coroutine

```
_Coroutine FmtLines {
    char ch;                              // used for communication
    int g, b;                             // global because used in destructor
    void main() {
        for ( ;; ) {                      // for as many characters
            for ( g = 0; g < 5; g += 1 ) { // groups of 5 blocks
                for ( b = 0; b < 4; b += 1 ) { // blocks of 4 characters
                    suspend();
                    cout << ch;   // print character
                }
                cout << "  ";             // block separator
            }
            cout << endl;                 // group separator
        }
    }
  public:
    FmtLines() { resume(); }              // start coroutine
    ~FmtLines() { if ( g != 0 || b != 0 ) cout << endl; }
    void prt( char ch ) { FmtLines::ch = ch; resume(); }
};
```

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

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



2.1.3 Correct Coroutine Usage

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

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 the “Zen” of the coroutine; 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) {
          case 1:
            fn = 0;  fn1 = fn;
            state = 2;
            break;
          case 2:
            fn = 1;  fn2 = fn1;  fn1 = fn;
            state = 3;
            break;
          case 3:
            fn = fn1 + fn2;  fn2 = fn1;  fn1 = fn;
            break;
        }
        suspend();
    }
}
```

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

- None of the coroutine's capabilities are used, and the program structure is lost in **switch** statement.
- Must do more than just *activate* the coroutine main to demonstrate an understanding of retaining data and execution state within a coroutine.

2.1.4 Device Driver

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

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

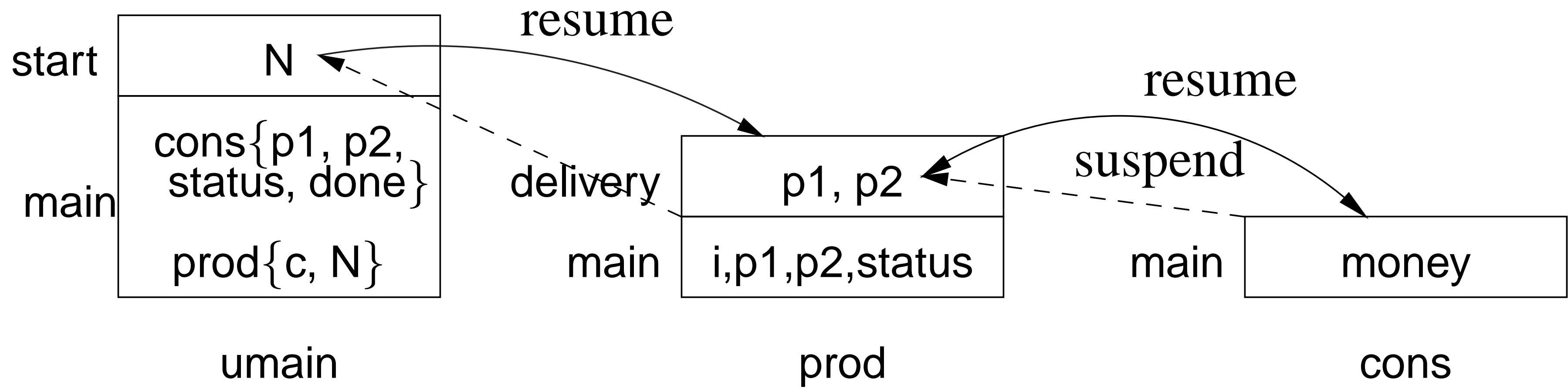
```
_Coroutine SerialDriver {
    unsigned char byte;
    int status;
    unsigned char *msg;
  public:
    driver( unsigned char *msg ) : msg( msg ) { resume(); }
    int next( unsigned char b ) {        // called by interrupt handler
        byte = b;
        resume();
        return status;
    }
  private:
    void main() {
      newmsg:
        for ( ;; ) {                      // parse messages
            status = CONT;
            int lnth = 0, sum = 0;
            do {
                suspend();
            } while ( byte != STX )       // look for start of message
          eomsg:
            for ( ;; ) {
                suspend();                // parse message bytes
                switch ( byte ) {
```

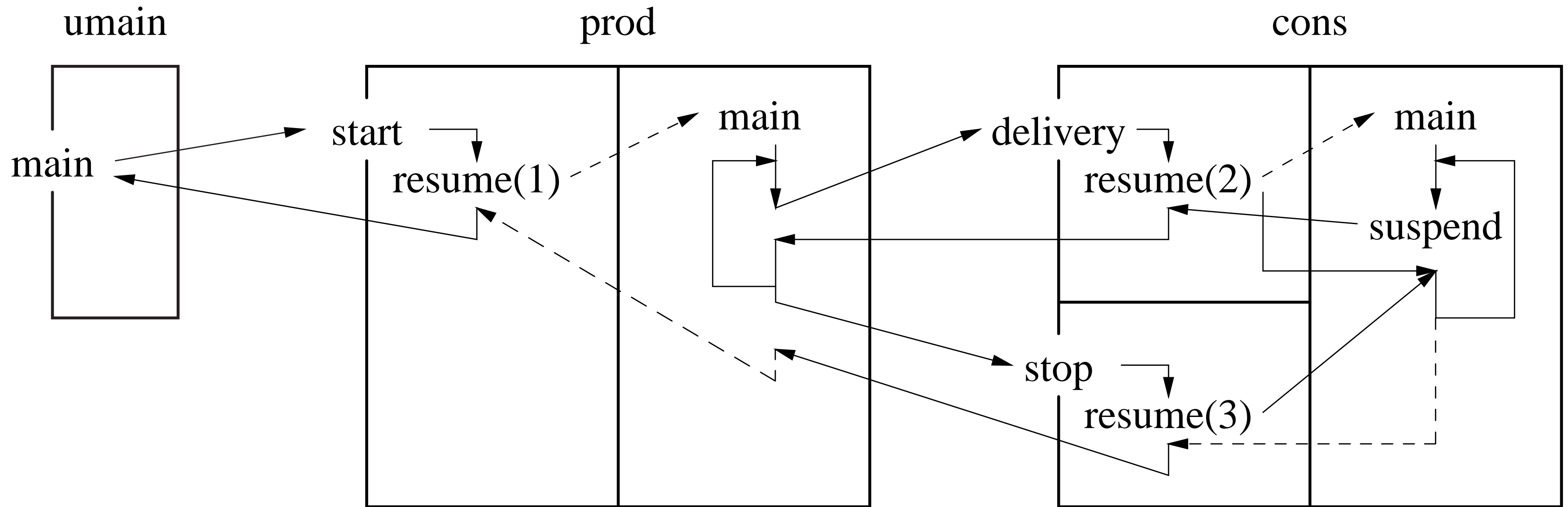
```
                    case STX:             // protocol violation
                        status = ERROR;
                        continue newMsg; // uC++ labelled continue
                    case ETX:             // end of message
                        break eomsg;      // uC++ labelled break
                    case ESC:             // escape next character
                        suspend();        // get escaped character
                        break;
                } // switch
                msg[lnth] = byte;         // store message
                lnth += 1;
                sum += byte;              // compute CRC
            } // for
            suspend();                    // obtain 1st CRC byte
            int crc = byte;
            suspend();                    // obtain 2nd CRC byte
            crc = (crc << 8) | byte;
            status = crc == sum ? MSG : ERROR;
            msgcomplete( msg, lnth ); // return message to OS
        } // for
    } // main
}; // SerialDriver
```

2.1.5 Producer-Consumer

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

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

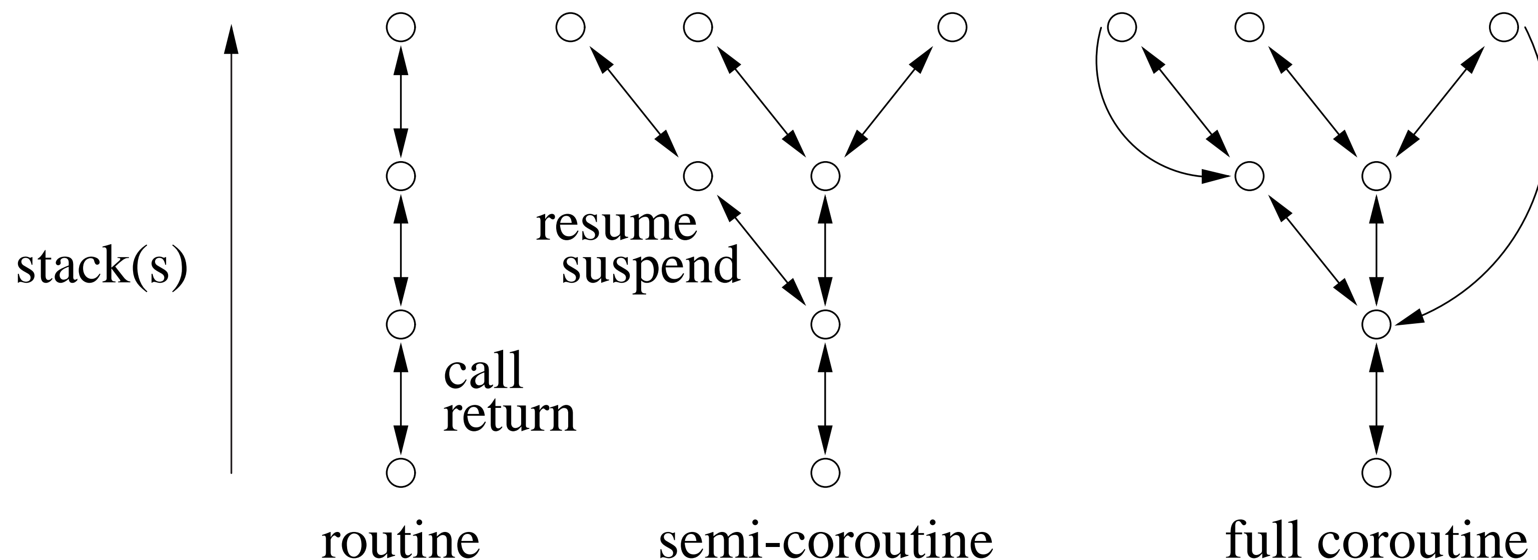

resume
start
N
resume
cons{p1, p2, status, done}
main
delivery
p1, p2
suspend
prod{c, N}
main
i,p1,p2,status
main
money
umain
prod
cons


umain
prod
cons
main
start
resume(1)
main
delivery
resume(2)
main
suspend
stop
resume(3)

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

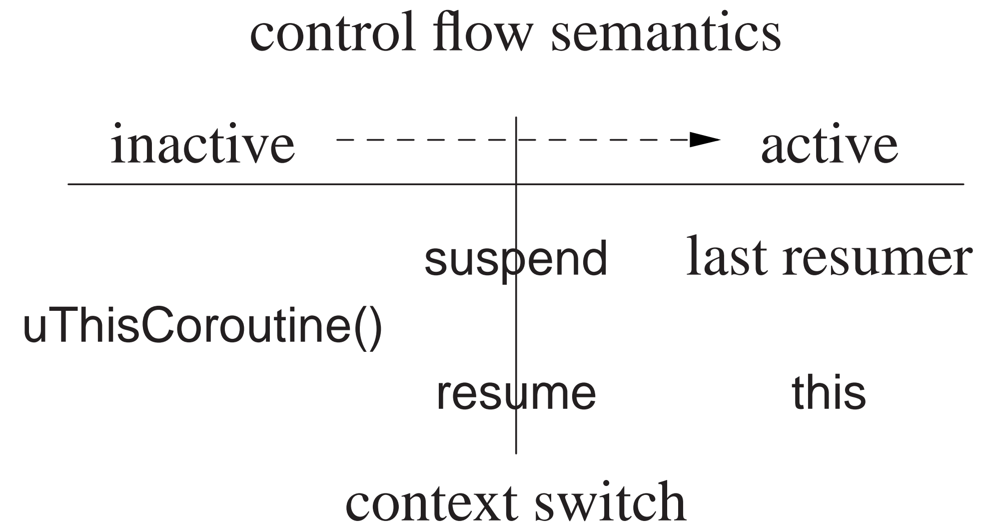
2.2 Full Coroutines

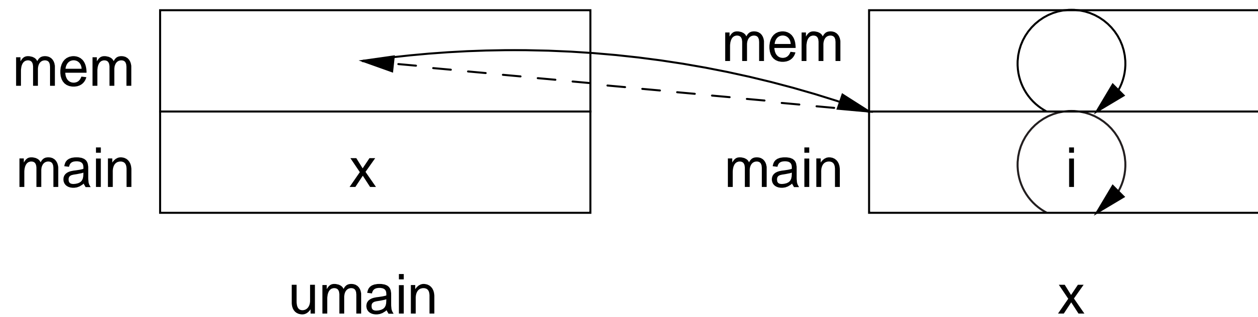
- **Semi-coroutine** activates the member routine that activated it.
- **Full coroutine** has a resume cycle; semi-coroutine does not form a resume cycle.

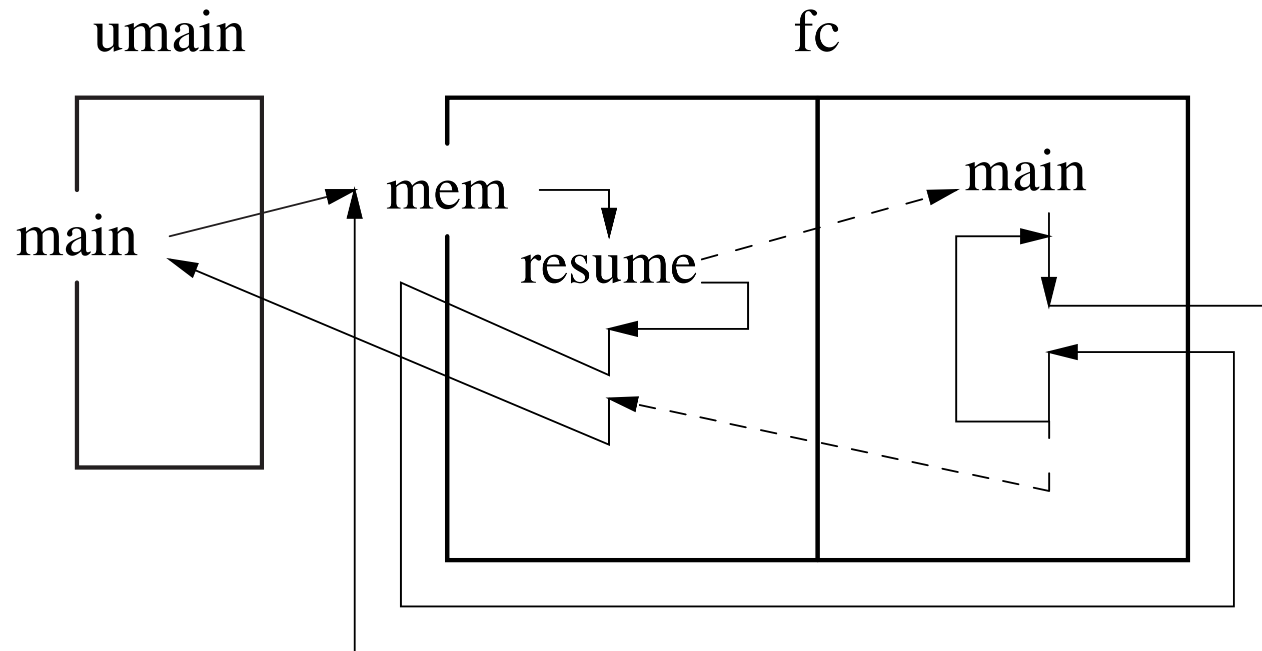


- A full coroutine is allowed to perform semi-coroutine operations because it subsumes the notion of semi-routine.

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







- 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:

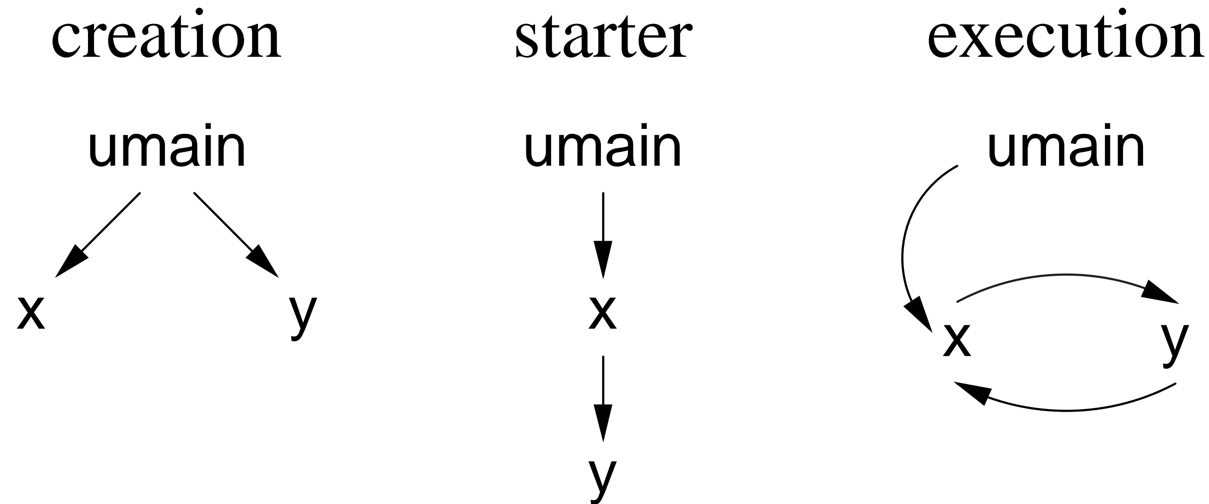
```
fc x(y), y(x);
```

- 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.

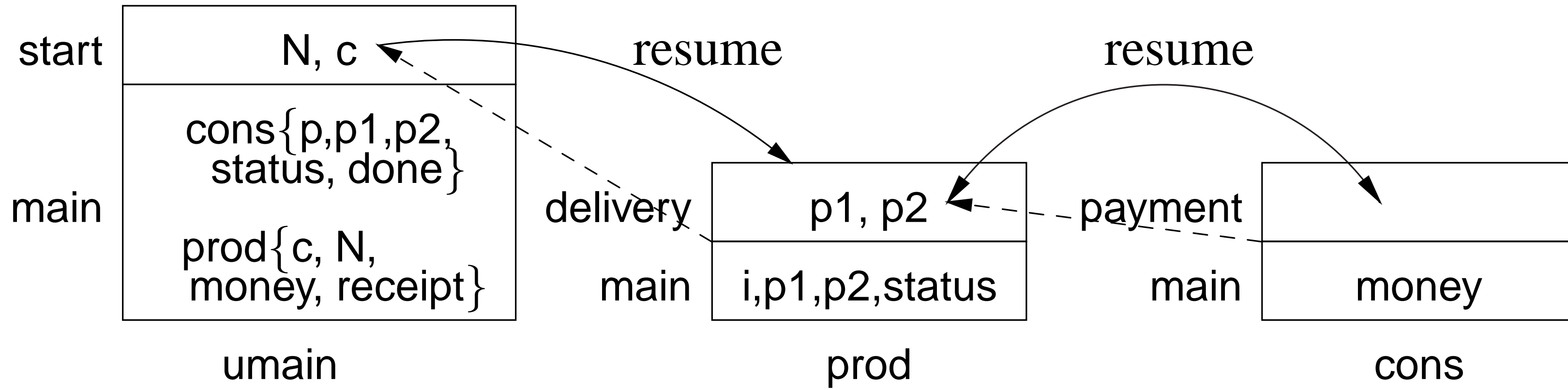
- In many cases, it is unnecessary to terminate all coroutines, just delete them.
- But it is necessary to activate uMain for the program to finish (unless exit is used).



2.2.1 Producer-Consumer

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

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



umain
prod
cons
main
start
resume(1)
main
delivery
resume(2)
main
payment
resume(3)
stop
resume(4)

2.3 Nonlocal Exceptions

- Exception handling is based on traversing a call stack.
- With coroutines there are multiple call stacks.
- Nonlocal exceptions move from one coroutine stack to another.

  _Throw [*throwable-event* [**_At** *coroutine-id*]] ;

- Hence, exceptions can be handled locally within a coroutine or nonlocally among coroutines.
- Local exceptions within a coroutine are the same as for exceptions within a routine/class, with one nonlocal difference:
  - An unhandled exception raised by a coroutine raises a nonlocal exception of type uBaseCoroutine::UnhandledException at the coroutine's last resumer and then terminates the coroutine.

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

- Call to c.mem resumes coroutine c and then coroutine c throws exception E but does not handle it.
- When the base of c's stack is reached, an exception of type uBaseCoroutine::UnhandledException is raised at uMain, since it last resumed c.
- *The original exception's (*E*) default terminate routine is not called because it has been caught and transformed.*
- *The coroutine terminates but control returns to its last resumer rather than its starter.*

```
_Coroutine C {
    void main() {
        for ( int i = 0; i < 5; i += 1 ) {
            try {
                _Enable {  // allow nonlocal exceptions
                    . . . suspend(); . . .
                }
            } catch( E ) { . . . }
        }
    }
  public:
    C() { resume(); } // prime loop
    void mem() { resume(); }
};
void uMain::main() {
    C c;
    for ( int i = 0; i < 5; i += 1 ) {
        _Throw E() _At c; // exception pending
        c.mem(); // trigger exception
    }
}
```

- **Nonlocal delivery is initially disabled for a coroutine**, so handlers can be set up before any exception can be delivered .
- Hence, nonlocal exceptions must be explicitly enabled before delivery can occur with **_Enable**.
- μC++ allows dynamic enabling and disabling of nonlocal event delivery.

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

- Specifying no exceptions enables/disables all nonlocal exceptions.
- **_Enable** and **_Disable** blocks can be nested, turning delivery on/off on entry and reestablishing the delivery state to its prior value on exit.
- The source coroutine delivers the nonlocal exception immediately but does not propagate it; propagation only occurs when the faulting coroutine becomes active.

  ⇒ must call one of the faulting coroutine's members that does a resume.