High Performance Computing

μC++

Fernando R. Rannou
Departamento de Ingenieria Informatica
Universidad de Santiago de Chile

September 22, 2023

Tipos Mutex

Un tipo mutex (clase mutex) consiste en un conjunto de variables y un conjunto de miembros mutex (métodos mutex) que operan sobre dichas variables. Un tipo mutex tiene al menos un método mutex

Objetos instanciados de tipos mutex poseen la propiedad que sus métodos mutex se ejecutan con exclusión mútua, es decir sólo una tarea puede estar ejecutándose en el método.

```
_Mutex  M {
private:
     char z(...);    // por defecto es nomutex
public:
     M();            // siempre nomutex
     ~M();           // siempre mutex
     int f1(...);    // por defecto mutex
     float f2(...); // por defecto mutex
}:
```

La ejecución de los miembros `f1()` y `f2()` es exclusiva.

Reglas para tipos mutex

La siguientes reglas aplican a objetos tipo mutex

1. Los miembros privados y protegidos son por defecto **_Nomutex**
2. Los miembros públicos son por defecto **_Mutex**
3. El comportamiento por defecto se puede cambiar usando explicitamente **_Mutex** y **_Nomutex**
4. El destructor siempre es **_Mutex** y no se puede cambiar
5. Cuando la ejecución de un miembro mutex comienza, el objeto mutex se *cierra* (lock), y cuando termina la ejecución se *abre*
6. Si una tarea invoca un miembro mutex cuando éste está cerrado, la tarea invocadora se bloquea
7. Una tarea dentro de un miembro mutex puede invocar otros miembros mutex, ya sea del mismo objeto u otros objetos mutex
8. Si una tarea tiene *cerrados* varios objetos mutex, se dice que es *dueña* de los locks de dichos objetos
9. El lock sobre un objeto mutex se libera cuando la tarea dueña del lock sale del objeto mutex o se bloquea dentro de él

Ejemplo: contador “atómico”

```
_Mutex atomiccounter {
    int cnt;
public:
    atomiccounter() { cnt = 0; }
    inc() { cnt += 1; }
};
```

Corutina

- es un objeto con su propio **estado de ejecución**
- no posee **hebra de ejecución**
- sus miembros no poseen **exclusión mutua**

Por lo tanto

- La ejecución de una corutina puede suspenderse y reanudarse, repetidamente
- Se necesita de una hebra que ejecute sus miembros
- La hebra, al entrar a la corutina, hace cambio de contexto, es decir, abandona el estado que trae y toma el último estado de la corutina
- Sería un error programático que varias hebras ejecuten la corutina al mismo tiempo

La corutina sirve para implementar problemas donde es necesario mantener el estado, entre invocaciones. Por ejemplo, una máquina de estados finitos (autómata)

Corutina

```
_Coroutine  M {
private:
    void main();
public:
    M();
    ~M();
    int  f(...);
};
```

- Toda corutina tiene un método privado o protegido llamado `main()`
- `main()` se *activa* a través de alguno de los métodos públicos de la corutina
- La ejecución se puede suspender con `suspend()` y reanudar con `resume()`

Existen dos tipo de corutines:

1. **Semi-corutina**: siempre reactiva la corutina que previamente había activado a la corutina
2. **Full-corutina**: activa explícitamente un miembro de otra corutina, no necesariamente la que la había activado

Semi-corutina

Implementemos el cálculo de Fibonacci con una semi-corutina

```
#include <uC++.h>
_Coroutine fibonacci {
   int fn;
   void main() {
     int fn1, fn2;
     fn = 1; fn1 = fn;
     suspend();//return to last resume
     fn = 1; fn2 = fn1; fn1 = fn;
     suspend();//return to last resume
     for ( ;; ) {
        fn = fn1 + fn2;
        suspend();//return to last resume
        fn2 = fn1; fn1 = fn;
     }
  }
public:
  int next() {
      resume();//transfer to last suspend
      return fn;
  }
};
```

```
void uMain::main() {
     fibonacci f1, f2;
     for (int i=1; i <= 6; i++) {
        cout << f1.next() << " " <<
                f2.next() << endl;
     }
}
$ u++ -o exe semi-coroutine1.cc
```

Primer `resume` en `next()` activa `main()`

Los `suspend` suspende el estado de la corutina y retornan al último `resume`
Posteriores `resume` en `next()` re-activan la corutina en el último `suspend`

Salida formateada con Semi-corutina

El objetivo de este programa es formatear los caracteres de entradas en 5 bloques de 4 caracteres cada uno. Ejemplo:

```
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

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

Salida formateada, cont.

```
_Coroutine FormatInput {
    char ch;
    int g, b;
    void main() {
        for (;;) {
            for (g=0; g<5; g++) {
                for (b=0; b<4; b++) {
                    suspend();
                    cout << ch;
                }
                cout << " ";
            }
            cout << endl;
        }
    }
public:
    FormatInput() { resume(); }
    ~FormatInput() {
        if (g != 0 || b != 0)
            cout << endl;
    }
    void prt(char ch) {
        FormatInput::ch = ch;
        resume();
    }
};
```

```
void uMain::main() {
    FormatInput fmt;
    char ch;
    for (;;) {
        cin >> ch;
        if (cin.fail()) break;
        fmt.prt(ch);
    }
}
```

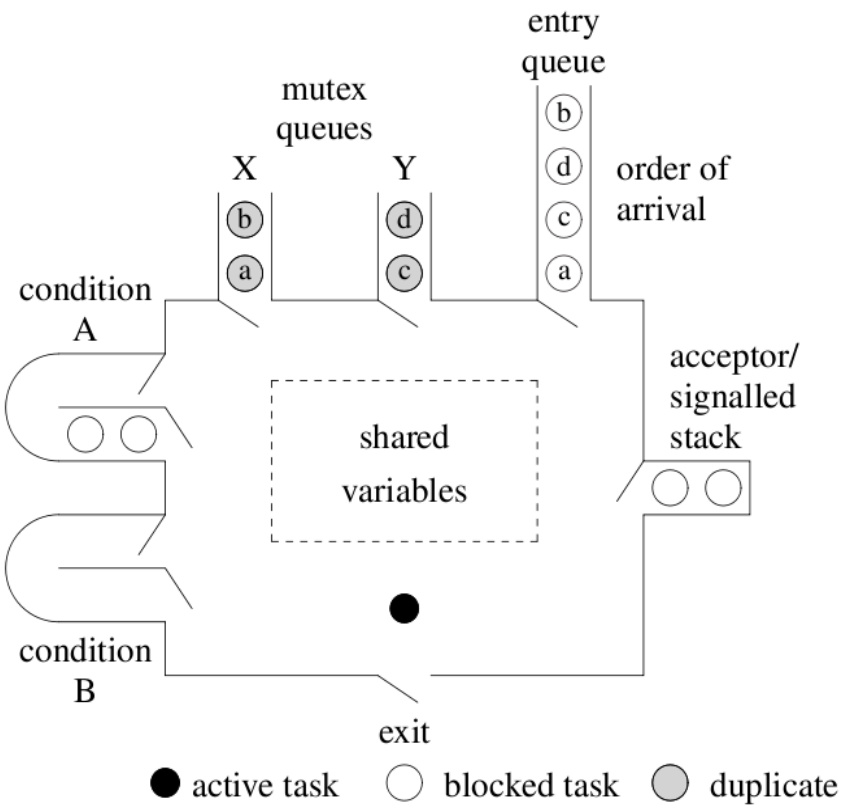
Monitor

- Un monitor es un tipo en el que todos los miembros públicos son por defecto **_Mutex**.
- Se puede establecer explícitamente un miembro público no mutex con **_Nomutex**
- El destructor siempre es **_Mutex**
- Entrada recursiva al monitor es permitido (*owner mutex lock*)

```
_Monitor  M {
private:
    int i;
public:
    M();
    ~M();
    void x(...);
    void y(...);
    _Nomutex int f();
};

uMain::main() {
    M m, ma[3], *mp;
    mp = new M;
}
```

Estructura objeto monitor



- Cada miembro mutex tiene una cola donde las tareas se bloquean cuando invocan dicho miembro mutex y el mutex está cerrado
- La cola de entrada mantiene el orden de las invocaciones al objeto
- Para que las tareas se bloqueen dentro del monitor, se declaran variables de condición

Ejemplo simple monitor

Implementemos un contador up-down

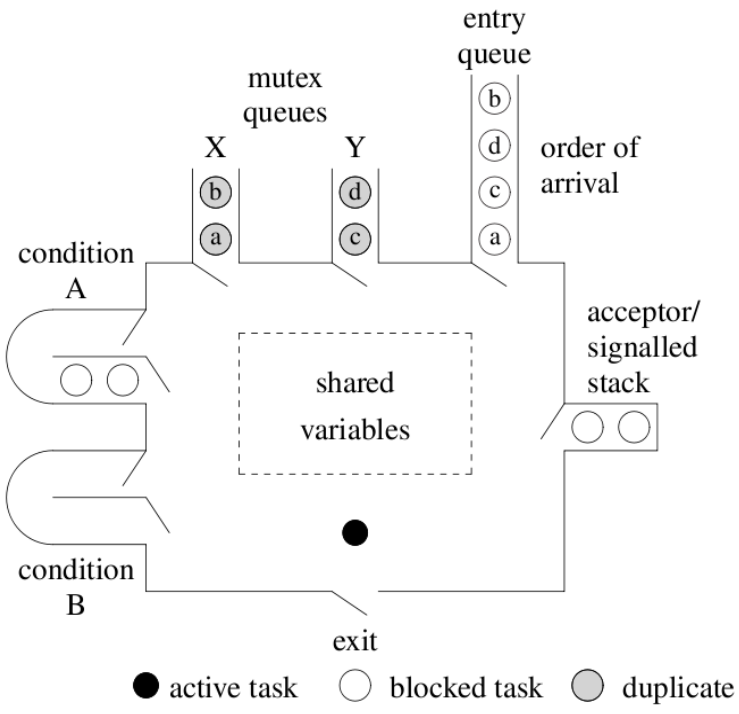
```
_Monitor UpDownCounter {
        int counter;
public:
    UpDownCounter() : counter(0) {}
    UpDownCounter(int init) : counter(init) {}
    int up()   { counter += 1; return counter; }
    int down() { counter -= 1; return counter; }
};

void uMain::main()
{
   UpDownCounter a(0), b(2), c;
   ...a.up();...
   ...b.down();...
   ...
   delete b;
}
```

Scheduling en monitores

Existen dos mecanismos de scheduling en los monitores:

1. **Scheduling interno**: se refiere a la planificación de **tareas** que están bloqueadas dentro del monitor. Este mecanismo utiliza variables de condición.
2. **Scheduling externo**: se refiere a la planificación de **tareas** que están bloqueadas fuera, en miembros mutex, del monitor, es decir tareas que solicitaron ingresar al monitor y se bloquearon.



Scheduling interno y variables de condición

Este tipo de scheduling, planifica tareas que se han bloqueado dentro del monitor en variables de condición

```
class uCondition {
public:
      void wait();
      void wait( long int info );
      void signal();
      void signalBlock();
      bool empty() const;
      long int front() const;
};
uCondition BufferVacio;
```

- `wait()` bloquea a la tarea invocadora en la cola de la variable de condición; se libera el mutex del monitor y se realiza scheduling interno
- `signal()` mueve la próxima tarea bloqueada en la cola de la variable de condición al tope del stack de aceptación. La tarea invocadora continúa su ejecución. Cuando sale del mointor o se bloquea dentro de él, el scheduler planifica la tarea del tope del stack

Variables de condición

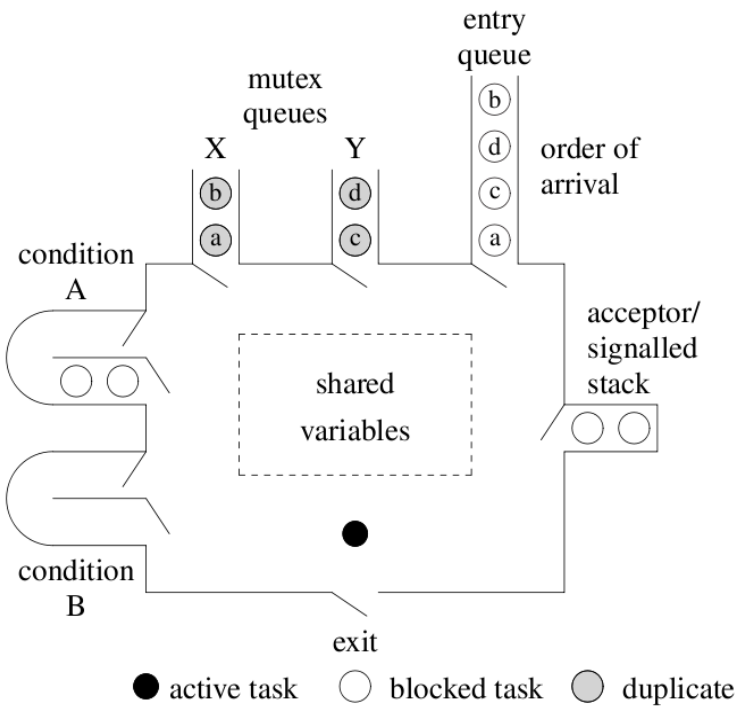
- `signalBlock()` bloquea la tarea invocadora en el stack y re-activa la próxima tarea en la cola de la variable de condición
- `empty()` retorna verdadero si no hay tareas bloqueadas en la variable de condición
- `front()` retorna el valor del entero almacenado en la tarea (durante un `wait(val)`) que está al frente de la cola

Ejemplo scheduling interno: Productor/Consumidor

```
_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;
      }
};
```

Scheduling externo en monitores

- Este tipo de scheduling se refiere a la planificación de tareas que están bloqueadas en las colas de los miembros mutex, es decir objetos que aún no han ingresado al monitor
- Scheduling externo es **explícito**, pues la planificación indica específicamente el miembro mutex del cual se debe planificar.
- La tarea que está dentro del monitor y *acepta*, se bloquea en el stack de aceptación/señalización
- Scheduling externo se lleva a cabo mediante la instrucción _**Accept**



_Accept y _When

La forma más simple de scheduling externo es aceptar uno de los miembros mutex del monitor:

```
_When ( exp-condicional )     // opcional
    _Accept( nombre-miembro-mutex );
```

- Cuando la expresión condicional dentro de **_When** es verdadera, se acepta la próxima tarea (en orden FIFO) que está bloqueada en la cola del miembro mutex especificado en **_Accept**
- Si la expresión es falsa, la tarea actual continúa su ejecución, sin aceptar nada
- Cuando se acepta un miembro mutex, la tarea *aceptadora* se bloquea en el tope del stack de aceptación/señalización, y libera el mutex que tenía sobre el objeto
- Luego, la tarea aceptada se apodera del mutex del objeto y ejecuta el miembro correspondiente

_Accept y _When, cont

Es posible especificar más de un miembro en **_Accept**. Por ejemplo,

```
_When( length >= 0 && length <= N )
  _Accept( insert, remove );
```

- Cuando hay más de un miembro en la lista del accept, se prioriza de izquierda a derecha. Si no existen llamados pendientes a un miembro mutex, se pasa al próximo miembro de la derecha y así sucesivamente
- Si no existen tareas bloqueadas en ninguna de las colas de los miebros mutex especificados en **_Accept**, la tarea aceptadora se bloquea en el tope del stack, hasta que llegue una invocación a uno de los miembros
- También es posible invocar a una o más funciones en la expresión condicional

  ```
  _When( bufferNotfull() && bufferNotempty() )
    _Accept( insert, remove );
  ```

  pero ninguna de éstas se puede bloquear

Ejemplo scheduling externo: Productor/consumidor

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

_Accept y _When, cont

- Cuando la tarea que es aceptada sale del monitor, se libera el mutex y el scheduler interno (implícito), re-activa la última tarea que se bloqueó en el stack de aceptación
- Cuando dicha tarea se ejecuta retoma el mutex del objeto y continúa su ejecución
- Cuando una tarea sale del monitor y no hay tareas esperando en el stack de señalización, entoces el scheduler externo planifica la próxima tarea (orden FIFO) de la cola de entrada al monitor

Forma extendida de _**Accept** y _**When**

¿Cuál es la diferencia entre **if** y _**When**?

```
_When ( exp-cond )
    _Accept( mutex-member )
           statement
or _When ( exp-cond )
    _Accept( mutex-member )
           statement
or
...
...
_When ( exp-cond )
  else
          statement
```

```
_When ( c1 ) _Accept( f1 );
or _When ( c2 ) _Accept( f2 );
```

es distinto a

```
if ( c1 ) _Accept( f1 );
else if( c2 ) _Accept( f2 );
```

En el primer caso, si c1 y c2 son verdaderos, ambos miebros mutex se aceptarán; primero `f1` y luego `f2`

En cambio en el segundo caso, si c1 y c2 ambos son verdaderos, sólo `f1` es aceptado

Ejemplo DatingService, scheduling externo

Intercambio de información entre dos tareas

```
_Monitor DatingService {
     int GirlPhoneNo, BoyPhoneNo;
public:
     DatingService() : GirlPhone(-1), BoyPhone(-1) { }

     int Girl( int PhoneNo ) {
          GirlPhoneNo = PhoneNo;
          if ( BoyPhoneNo == -1 )
               _Accept( Boy );
          int temp = BoyPhoneNo;
          BoyPhoneNo = -1;
          return temp;
     }

     int Boy( int PhoneNo ) {
          BoyPhoneNo = PhoneNo;
          if ( GirlPhoneNo == -1 )
               _Accept( Girl );
          int temp = GirlPhoneNo;
          GirlPhoneNo = -1;
          return temp;
     }
};
```

Tareas

- Una tarea es un objeto con su propia hebra de control, estado de ejecución y cuyos miembros públicos poseen exclusión mútua

```
_Task T {
 private:
     ...
 protected:
     ...
     void main();
 public:
     ...
     void r(...);
};
```

```
T *tp;
{
     T t, ta[3];
     tp = new T;
     ...
     t.r(...);
     ta[1].r(...);
     tp->r(...);
}
delete tp;
```

- Toda tarea debe tener un miembro privado o protegido `main()`, en el cual la tarea comienza su ejecución al momento de creación/instanciación
- `main()` no tiene argumentos y no retorna valor
- Los clientes se comunican con la tarea traspasando información a través de los miembros públicos de la tarea

Tareas (cont)

- Cuando una hebra es creada, la hebra llamadora ejecuta el constructor
- Luego se crean el estado de ejecución y la hebra que comienza la ejecución de `main`
- La hebra llamadora retorna al punto donde invocó la creación de la tarea, y continúa su ejecución concurrentemente con la tarea recién creada
- Una tarea termina cuando termina la ejecución de `main()`. La hebra es eliminada, pero la tarea se convierte en un monitor, y por lo tanto se podría recuperar información a través de sus métodos públicos
- Una tarea se destruye cuando se invoca al destructor. El destructor puede ser invocado implicitamente, cuando el bloque que contiene la declaración de la tarea termina, o explicitamente con un **delete**
- Como el destructor es siempre mutex, la tarea no es destruida hasta cuando no haya ninguna hebra ejecutándose dentro de la tarea

Ejemplo Productor/comsumidor con tareas, uMain

```
void uMain::main() {
     const int NoOfCons = 2, NoOfProds = 3;
     BoundedBuffer buf;           // Monitor
     Consumer *cons[NoOfCons];    // Tareas Consumidoras
     Producer *prods[NoOfProds]; // Tareas Productoras

     for ( int i = 0; i < NoOfCons; i += 1 )
          cons[i]  = new Consumer( buf );
     for ( int i = 0; i < NoOfProds; i += 1 )
          prods[i] = new Producer( buf );

     for ( int i = 0; i < NoOfProds; i += 1 )
          delete prods[i];

     for ( int i = 0; i < NoOfCons; i += 1 )
          buf.insert( -1 );
     for ( int i = 0; i < NoOfCons; i += 1 )
          delete cons[i];
}
```

Ejemplo Productor/Consumidor con tareas, productor

```
_Task Producer {
     BoundedBuffer &Buffer;
public:
     Producer( BoundedBuffer &buf ) : Buffer( buf ) {}
private:
     void main() {
          const int NoOfItems = rand() % 20;
          int item;

          for (int i = 1; i <= NoOfItems; i += 1) {
               yield( rand() % 20 );     // duerma un rato
               item = rand() % 100 + 1;
               Bufffer.insert( item );
          }
     }
};
```

Ejemplo Productor/Consumidor con tareas, consumidor

```
_Task Consumer {
     BoundedBuffer &Buffer;
public:
     consumer( BoundedBuffer &buf ) : Buffer( buf ) {}
private:
     void main() {
          int item;

          for ( ;; ) {
               item = Buffer.remove();
               if ( item == -1 ) break;
               yield( rand() % 20 );
          }
     }
};
```

El BoundeBuffer con tarea

- En todos los ejemplos anteriores hemos implementado el `BoundedBuffer` como un monitor, es decir como un objeto con EM, pero sin hebra de ejecución
- Ahora lo implementamos como una tarea
- Luego, habrán tareas productoras y tareas consumidoras que accesarán al monitor, además de la tarea que administra el monitor

BoundedBuffer como tarea

```
_Task BoundedBuffer {
      int front, back, count, elements[20];
public:
      BoundedBuffer() : front(0), back(0), count(0) {}
      ~BoundedBuffer() {}
      _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;
      }
};
private:
      void main() {
            ...
      }
};
```

BoundedBuffer como tarea: el main

```
...
private:
     void main() {
          for (;;) {
               _Accept( ~BoundedBuffer )
                    break;
               or _When (count != 20) _Accept(insert);
               or _When (count !=  0) _Accept(remove);
          }
     }
};
```

- La tarea BoundedBuffer actúa como un simple orquestador, aceptando tareas productoras y consumidoras
- La tarea BoundedBuffer no inserta, ni elimina elementos del buffer
- Cuando BoundedBuffer acepta uno de sus miembros mutex, la hebra se bloquea en el stack de señalización/aceptación
- ¿Por qué se acepta el destructor?

Aceptando el destructor

- La técnica de aceptar al destructor es generalmente usada para terminar un objeto mutex.
- Aceptar el destructor en el `main` tiene una semántica distinta a aceptar otro miembro mutex

```
private:
 void main() {
      for (;;) {
         _Accept(~BoundedBuffer)
              break;
         or _When (count != 20)
             _Accept(insert);
         or _When (count != 0)
             _Accept(remove):
    }
 }
```

Cuando se invoca al destructor, el llamador se bloquea inmediatamente

Cuando el destructor es aceptado, el llamador es puesto en al stack de señalización, en vez del aceptador

El aceptador retorna del **_Accept** sin haber ejecutado el destructor y puede realizar tareas de cleanup antes de terminar

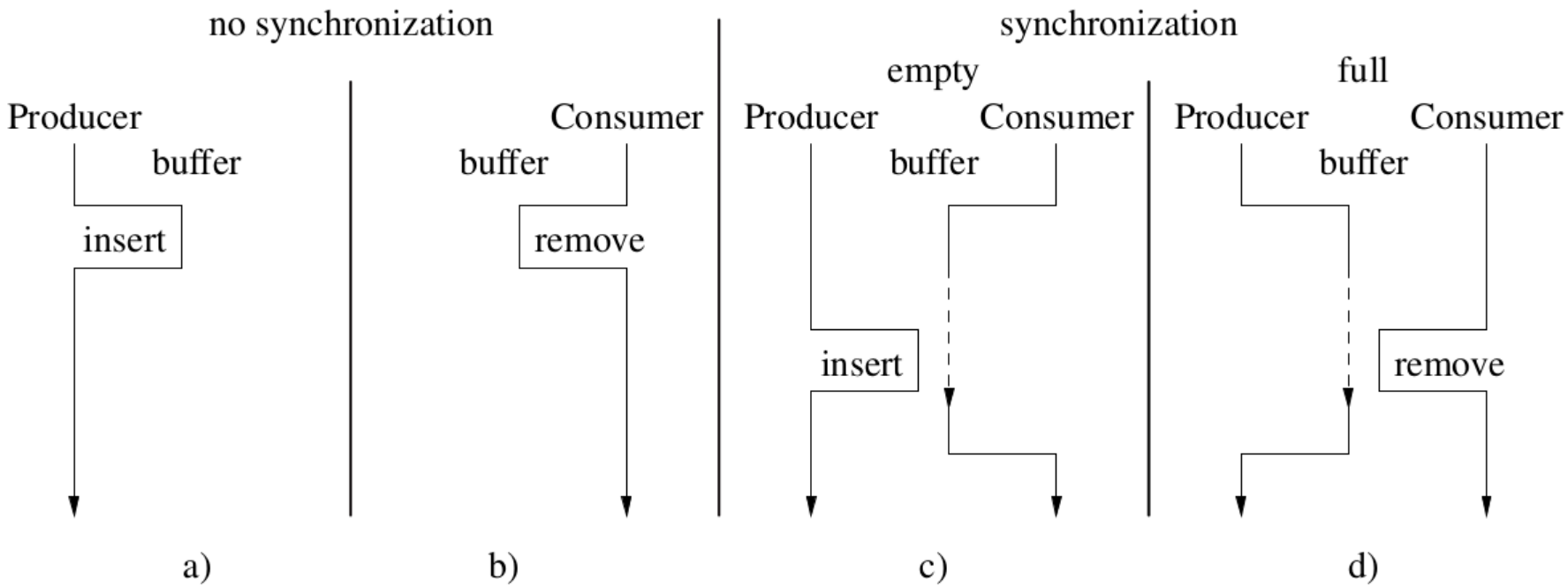
Cuando el main termina, la tarea se convierte en monitor y el llamador ejecuta el destructor

Comunicación directa e indirecta

- Un objeto con su propia hebra de control, EM, y estado de ejecución es llamado un **Objeto Activo (OA)**
- Los otros tipo de objetos son llamados **Objetos Inactivos (OI)**
- Hasta el momento hemos visto que dos OA pueden comunicarse, por ejemplo, a través de un OI como el monitor.
- En este sentido, existen dos formas de comunicación entre OA:
  1. **Directa**: Dos OA se comunican directamente enter sí
  2. **Indirecta**: Dos OA se comunican a través de un tercero, por ejemplo datos compartidos o algún OI

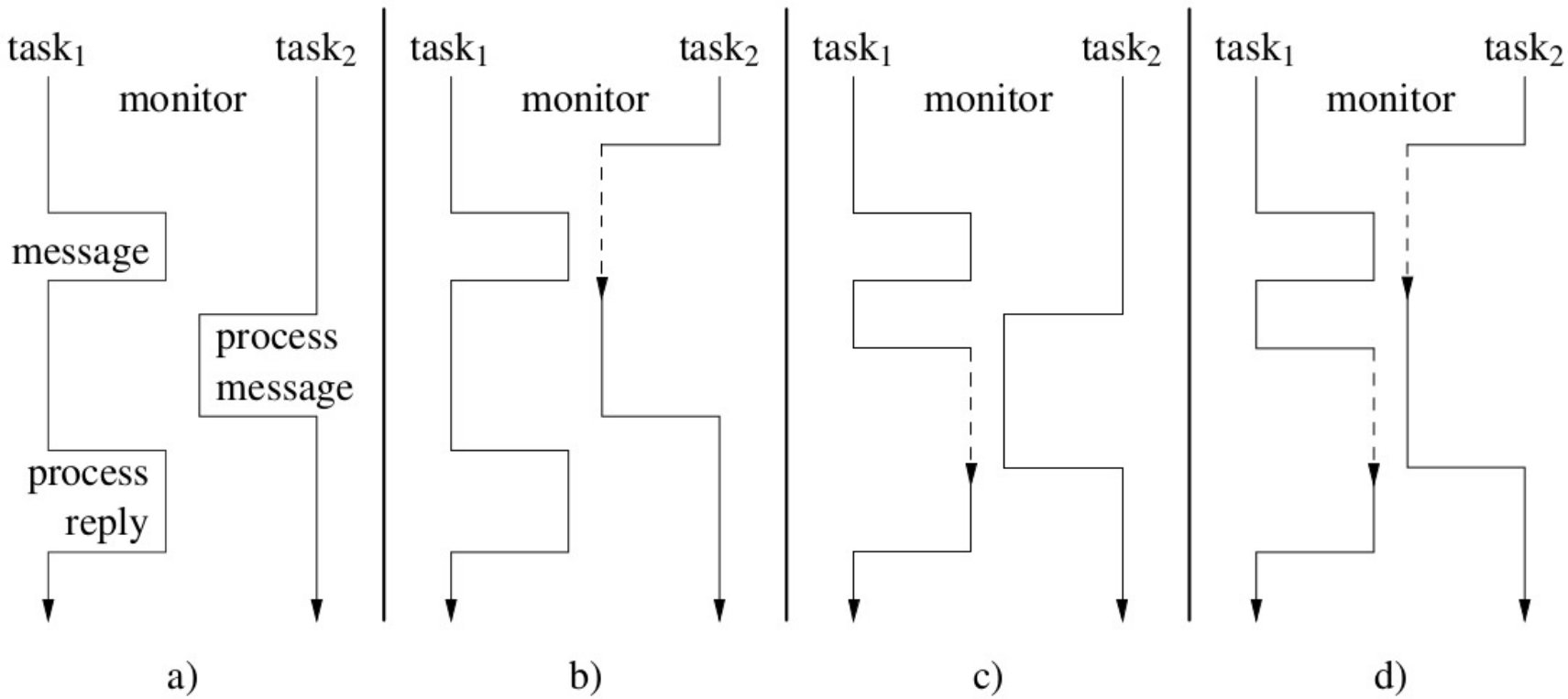
Comunicación indirecta unidireccional

- El `BoundeBuffer` es un ejemplo típico de OI que sirve para comunicar indirectamente dos o más OA.
- Si sólo hay un consumidor y un productor, y el buffer no está lleno ni vacío (a, b), los OA operan sin sincronización (aún hay EM)
- En c) y d), los OA se bloquean y necesitan cooperación del OA no bloqueado para continuar



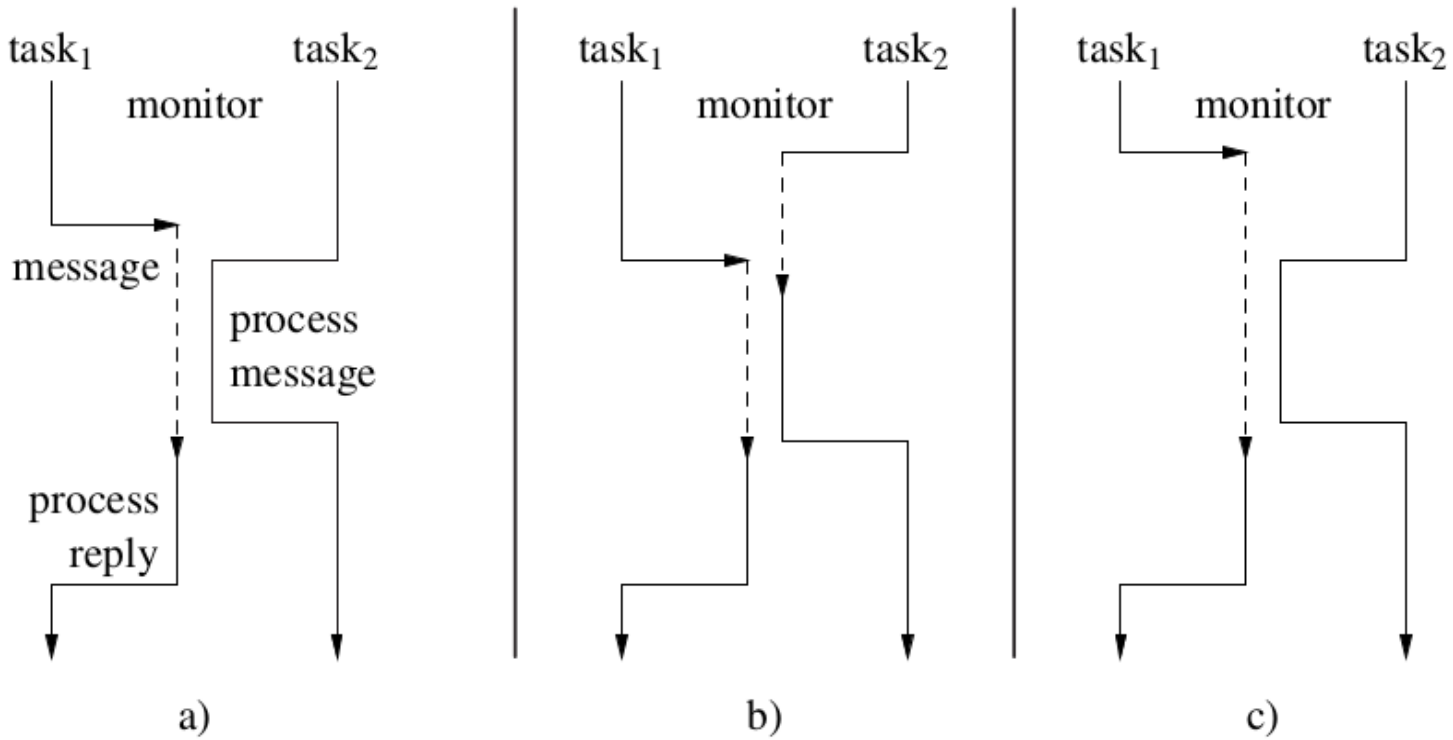
Comunicación indirecta bidireccional

- En el caso anterior, los datos fluyen en un sólo sentido, y el productor no necesita una respuesta del consumidor
- Cuando uno de los objetos necesita obtener una respuesta a su mensaje, decimos que la comunicación es bidireccional
- En el siguiente ejemplo, la tarea1 puede continuar su trabajo y chequear la respuesta en un tiempo posterior



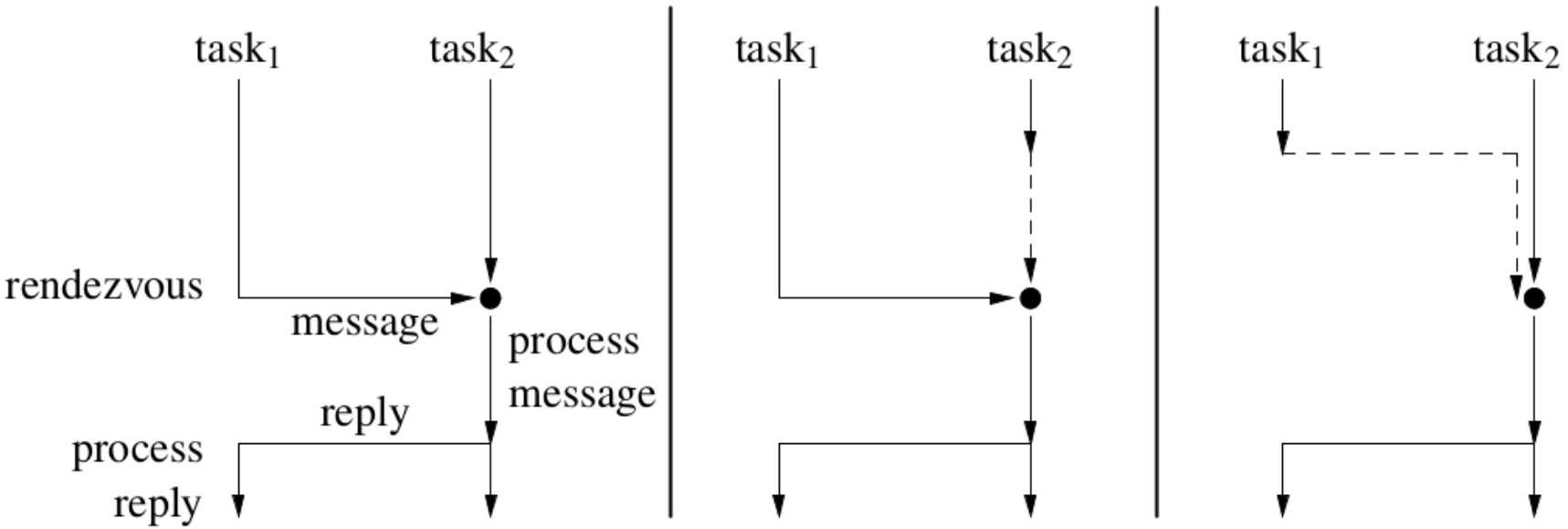
Comunicación indirecta bidireccional (cont)

- En este caso, la tarea1 no puede continuar si no recibe la respuesta de la tarea2



Comunicación directa

- Comunicación directa requiere un *rendezvous* entre los OA. Es decir, primero deben sincronizarse antes de transferir información
- En a) ambos OA llegan al punto de sincronización al mismo tiempo. La tarea1 entrega el mensaje y luego la tarea2 la procesa
- En uC++, en realidad es la tarea1 la que procesa el mensaje, mientras la tarea2 se bloquea
- En b) y c) una de las tareas llega antes que la otra y se bloquea
- ¿Cuáles son las ventajas y desventajas que sea la tarea1 o la tarea2 la que procese el mensaje?



Ejemplo comunicación directa

Este ejemplo implementa el productor/consumidor como un par de tareas. Posee varias restricciones y es bastante secuencial. El propósito es mostrar cómo dos tareas podría comunicarse sin usar un monitor

```
_Task Producer;

_Task Consumer {
      uCondition check;
      Producer &prod;
      int p1, p2, status;
public:
      Consumer(Producer &p) :
            prod(p), status(0) {}
      void stop() {}
      int delivery(int p1, int p2);
private:
      void main();
};
```

```
_Task Producer {
      Consumer *cons;
      int N, money, receipt;
public:
      Producer() : receipt(0) {}
      ~Producer();
      int payment(int money);
      void start(int N, Consumer &c);
private:
      void main();
};

void uMain::main() {
      Producer prod;
      Consumer cons(prod);
      prod.start(5, cons);
}
```

Productor/Consumidor con tareas

```
void Producer::main() {
  int i, p1, p2, status;
  _Accept( start );
  for (i = 1; i <= N; i += 1) {
      p1 = rand() % 100;
      p2 = rand() % 100;
      cout << "prod␣delivers" <<
          p1 << "␣␣" << p2 << endl;
      status = cons->delivery(p1,p2 );
      _Accept( payment );
      cout << "prod␣status␣" <<
          status << endl;
  }
  cons->stop();
  cout << "prod␣stops" << endl;
}
```

```
void Producer::start(int N,
                     Consumer &c) {
     Producer::N = N;
     cons = &c;
}

Producer::~Producer() {}

int Producer::payment( int money) {
   Producer::money = money;
   cout << "prod␣payment␣of␣$" <<
        money << endl;
   receipt += 1;
   return receipt;
}
```

Productor/Consumidor con tareas

```
void Consumer::main() {
  int money = 1, receipt;
  for (;;) {
     _Accept(stop) {
       break;
     } or _Accept(delivery) {
        cout << "cons␣receives:␣" <<
                p1 << "," << p2 << endl;
        status +=1;
        check.signalBlock();
        cout << "␣and␣pays␣$" <<
                          money << endl;
        receipt = prod.payment(money);
        cout << "cons␣receipt␣#" <<
                        receipt << endl;
        money += 1;
    }
  }
  cout << "cons␣stops␣" << endl;
}
```

```
int Consumer::delivery(int p1, int p2) {
  Consumer::p1 = p1;
  Consumer::p2 = p2;
  check.wait();
  return status;
}
```

Incrementando concurrencia

El ejemplo anterior muestra una forma sencilla para aumentar la concurrencia entre la tarea llamadora y la aceptadora. El BoundedBuffer lo podemos reescribir:

```
_Task BoundedBuffer {
     int front, back, count, elements[20];
public:
     BoundedBuffer() : front(0), back(0), count(0) {}
     ~BoundedBuffer() {}
     _Nomutex int query() { return count; }
     void insert(int elem) { elements[back] = elem; }
     int remove() { return elements[front]; }
};

protected:
 void main() {
     for (;;) {
          _Accept( ~BoundeBuffer )
               break;
          or _When (count != 20 ) _Accept(insert)
          {
               back = (back+1)% 20;
               count += 1;
          } or _When (count != 0)  _Accept(remove)
          {
               front = (front+1)%20;
               count -= 1;
          }
```

Sincronización explícita

Hasta ahora hemos visto mecanismos implícitos para proveer:

1. Exclusión mútua: _**Mutex**, _**Monitor**, _**Comonitor**, y _**Task**
2. Sincronización: _**Accept**, `wait`, `signal` y `signalblock`

Estos mecanismos son suficientes para implementar aplicaciones altamente concurrentes

Sin embargo, en casos especiales es posible que el programador necesite de mecanismos explícitos

Entre los mecanismos de sincronización de bajo nivel, uC++ provee

- Semáforos contadores
- Tres tipos de locks
- Barreras

Semáforos contadores

Un semáforo contador en uC++ consiste de un contador y de una cola de tareas que se bloquean en el semáforo

```
#include <uSemaphore.h>

class uSemaphore {
public:
  uSemaphore(unsigned int count=1);
  void P();
  bool P( uDuration duration );
  bool P( uTime time );
  bool TryP();
  void V( unsigned int times = 1 );
  int counter() const;
  bool empty() const;
};
uSemaphore x, y(0), *z;
z = new uSemaphore(4);
```

`P()` decrementa el contador. Si el valor es mayor o igual que cero retorna. Si no, la tarea llamadora se bloquea

`V()` incrementa el semáforo y si hay tareas bloquedas, reactiva una en orden FIFO

`TryP()` retorna verdadero si el semáforo está tomado. Nunca se bloquea

Semáforos (cont)

Suponga que tres tareas deben ejecutar una sección de código en un cierto orden. `S2` y `S3` sólo se pueden ejecutar después que `S1` haya terminado.

La versión `V( int )` nos sirve en este caso

```
void uMain::main() {
     // comienza cerrado
     uSemaphore s(0);

     // pasamos el semaforo como
     // referencia a las tareas
     X x(s);
     Y y(s);
     Z z(s);
}
```

```
_Task X {...};
_Task Y {...};
_Task Z {...};

X::main() {
     s.P();
     S2();
}

Y::main() {
     S.P();
     S3();
}

Z::main() {
     S1();
     // en vez de dos s.V();
     s.V(2);
}
```

Spin locks

Un spin lock es implementado con busy-waiting

En uC++ existen dos tipos de spin locks

1. Apropiativo: **uSpinLock**
   Una vez que el lock es bloqueado por una tarea, ninguna otra tarea puede ejecutarse en el procesador donde se ejecuta la tarea

   Sólo puede usarse para EM
2. No apropiativo: **uLock**
   No tiene la restricción anterior

   Puede usarse para EM y sincronización

Un lock está abierto (1) o cerrado (0)

Locks no garantizan un orden en particular en que las tareas toman el lock. En teoria podrían producir deadlock

Spin locks (cont)

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

uLock x, y, *z;
z = new uLock( 0 );
```

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

Si el lock está abierto, `acquire()` lo cierra y retorna. Si está cerrado, la tarea queda en busy-waiting

`release()` abre el lock

Note que con `uLock` es posible que una tarea tome un lock y otra lo libere

Con `uSpinLock` la única tarea que puede abrirlo es aquella que lo cerró

Ejemplo de locks en sincronización

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

```
_Task T2 {
  uLock &lk;
  void main() {
      ...
      S1();
      lk.release();
};:
public:
  T2( uLock &lk ) : lk(lk) {}
};
```

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

Ejemplo de locks en EM

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

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

Barreras

Una barrera permite que un grupo establecido de tareas se sincronicen en un punto dado del programa

En uC++ las barreras son implementadas con una corutina monitor

```
#include <uBarrier.h>

_Mutex _Coroutine uBarrier {
 protected:
    void main() {
        for ( ;; ) {
            suspend();
        }
    }
 public:
    uBarrier( unsigned int total );
    Nomutex unsigned int total() const;
    Nomutex unsigned int waiters() const;
    void reset( unsigned int total );
    void block();
    virtual void last() {
        resume();
    }
};
uBarrier x(10), *y;
y = new uBarrier( 20 );
```

Ejemplo de barreras

```
X::main() {
      ...
      S1();
      b.block();
      S5();
}
```

```
Y::main() {
      ...
      S2();
      b.block();
      S6();
}
```

```
Z::main() {
      ...
      S3();
      b.block();
      S7();
}
```

```
void uMain::main() {
   uBarrier b( 3 );
   X x( b );
   Y y( b );
   Z z( b );
}
```

Owner locks

Un owner lock pertenece sólo a una tarea; la primera tarea que lo bloquea

Todas las otras tareas que intentan adquirir el lock se bloquean (no hay busy-waiting)

El dueño de un owner lock puede adquirir múltiples veces el lock pero para liberarlo completamente debe liberarlo el mismo número de veces

Por lo anterior, estos locks sólo se usan para EM

```
class uOwnerLock {
  public:
     uOwnerLock();
     unsigned int times() const;
     uBaseTask *owner() const;
     void acquire();
     bool tryacquire();
     void release();
};
uOwnerLock x, y, *z;
z = new uOwnerLock;
```

`times()` retorna el número de veces que el lock ha sido tomado
`owner()` retorna la dirección de la tarea que actualmente tiene el lock

Condition lock

Un condition lock es similar a una variable de condición, pero no está asociada a ningún monitor, sino que a un owner lock

En ese sentido son parecidos a las variables de condición de Pthreads

Condition lock sólo puede usarse para sincronización. Por qué?

```
class uCondLock {
 public:
   uCondLock();
   bool empty();
   void wait( uOwnerLock &lock );
   bool wait( uOwnerLock &lock, uDuration duration );
   bool wait( uOwnerLock &lock, uTime time );
   void signal();
   void broadcast();
};
uCondLock x, y, *z;
z = new uCondLock;
```

Un condition lock tiene asociada un cola donde se bloquean las tareas

`wait()` bloquea a la tarea llamadora en el condition lock y libera la tarea (FIFO) bloqueada en un owner lock (todo atómicamente)

`broadcast()` libera todas la tareas bloqueadas en el condition lock

Ejemplo condition lock

```
_Task T1 {
  uOwnerLock &olk;
  uCondLock  &clk;
  void main() {
      olk.acquire();
      if ( ! done )
          clk.wait( olk );
      olk.release();
      S2();
  }
public:
  T1( uOwnerLock &olk,
      uCondLock  &clk ) :
   olk(olk), clk(clk) {}
};
```

```
_Task T2 {
  uOwnerLock &olk;
  uCondLock  &clk;
  void main() {
      S1();
      olk.acquire();
      done = true;
      clk.signal();
      olk.release();
  }
public:
  T2( uOwnerLock &olk,
      uCondLockm &clk) ;
   olk(olk), clk(clk) {}
};
```

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