Monitores

Programación concurrente

Hasta ahora vimos:

- Los programas concurrentes pueden hace un uso eficiente de los recursos computacionales.

Hasta ahora vimos:

- Los programas concurrentes pueden hace un uso eficiente de los recursos computacionales.
- Sin embargo, en un modelo con memoria compartida *interleavings* indeseables pueden generar resultados erróneos.

Hasta ahora vimos:

- Los programas concurrentes pueden hace un uso eficiente de los recursos computacionales.
- Sin embargo, en un modelo con memoria compartida *interleavings* indeseables pueden generar resultados erróneos.
- Los *Semáforos* son una herramienta eficiente para resolver los problemas de sincronización.

Hasta ahora vimos:

- Los programas concurrentes pueden hace un uso eficiente de los recursos computacionales.
- Sin embargo, en un modelo con memoria compartida *interleavings* indeseables pueden generar resultados erróneos.
- Los *Semáforos* son una herramienta eficiente para resolver los problemas de sincronización.
    1. Son muy bajo nivel (es fácil olvidarse un acquire o un release).

Hasta ahora vimos:

- Los programas concurrentes pueden hace un uso eficiente de los recursos computacionales.
- Sin embargo, en un modelo con memoria compartida *interleavings* indeseables pueden generar resultados erróneos.
- Los *Semáforos* son una herramienta eficiente para resolver los problemas de sincronización.
    1. Son muy bajo nivel (es fácil olvidarse un acquire o un release).
    2. No están vinculados a datos (pueden aparecer en cualquier parte del código).

Monitores

Monitores

- Combina tipos de datos abstractos y exclusión mutua
  - Propuesto por Tony Hoare [1974]

Monitores

- Combina tipos de datos abstractos y exclusión mutua
  - Propuesto por Tony Hoare [1974]
- Incorporados en lenguajes de programación modernos
  - Java
  - C#

Elementos principales

Elementos principales

- Conjunto de operaciones encapsuladas en módulos.

Elementos principales

- Conjunto de operaciones encapsuladas en módulos.
- Un único *lock* que asegura exclusión mutua para todas las operaciones del monitor.

Elementos principales

- Conjunto de operaciones encapsuladas en módulos.
- Un único *lock* que asegura exclusión mutua para todas las operaciones del monitor.
- Variables especiales llamadas *condition variables*, utilizadas para programar sincronización condicional.

Ejemplo: Contador

```
monitor Contador {

  private int contador = 0;

  public void incrementar() {
    contador++;
  }

  public void decrementar() {
    contador--;
  }

}
```

Condition variables

Condition variables

- Están ligadas al monitor.

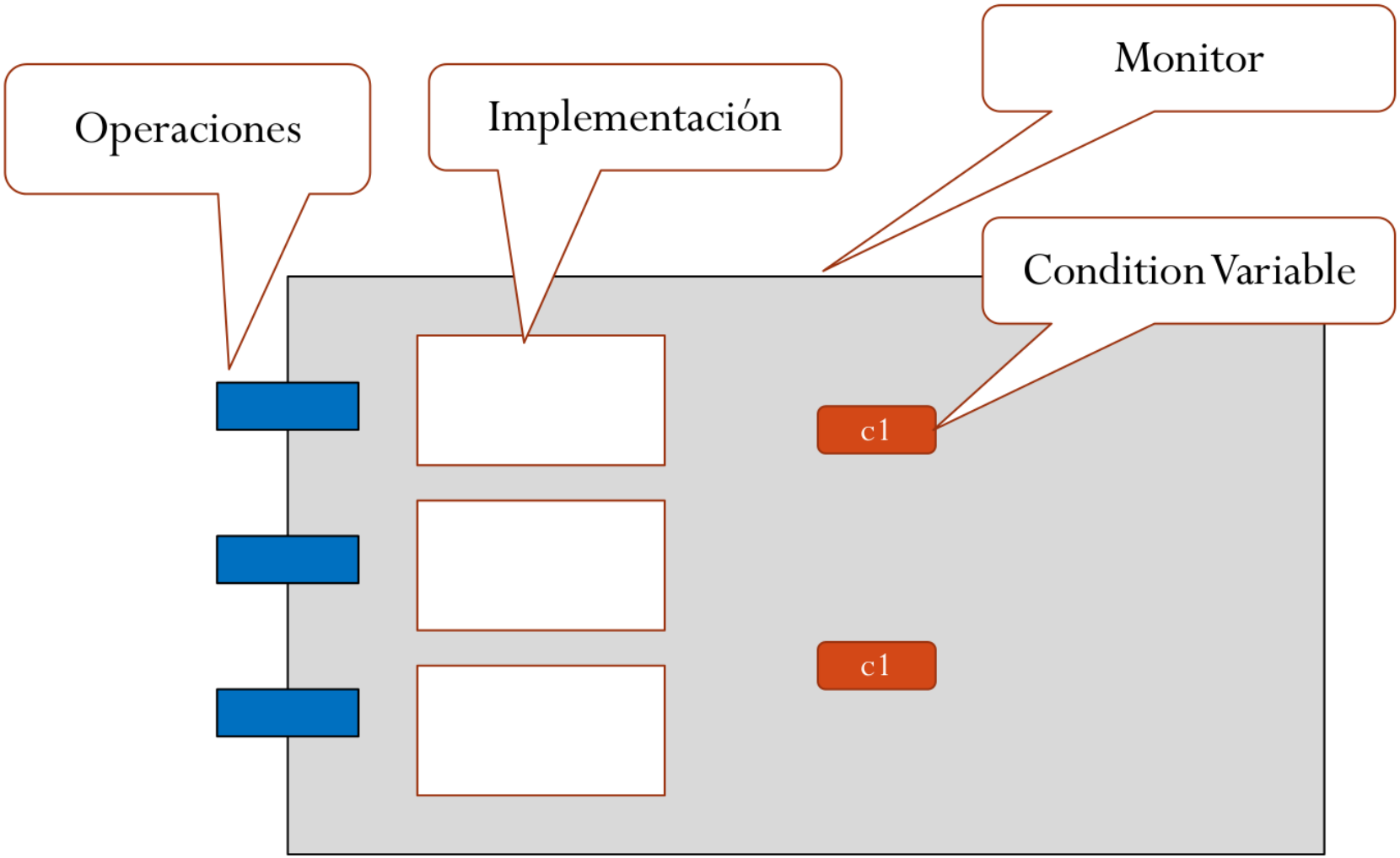
Condition variables

- Están ligadas al monitor.
- Poseen dos operaciones:
  - `wait()`
  - `notify()`

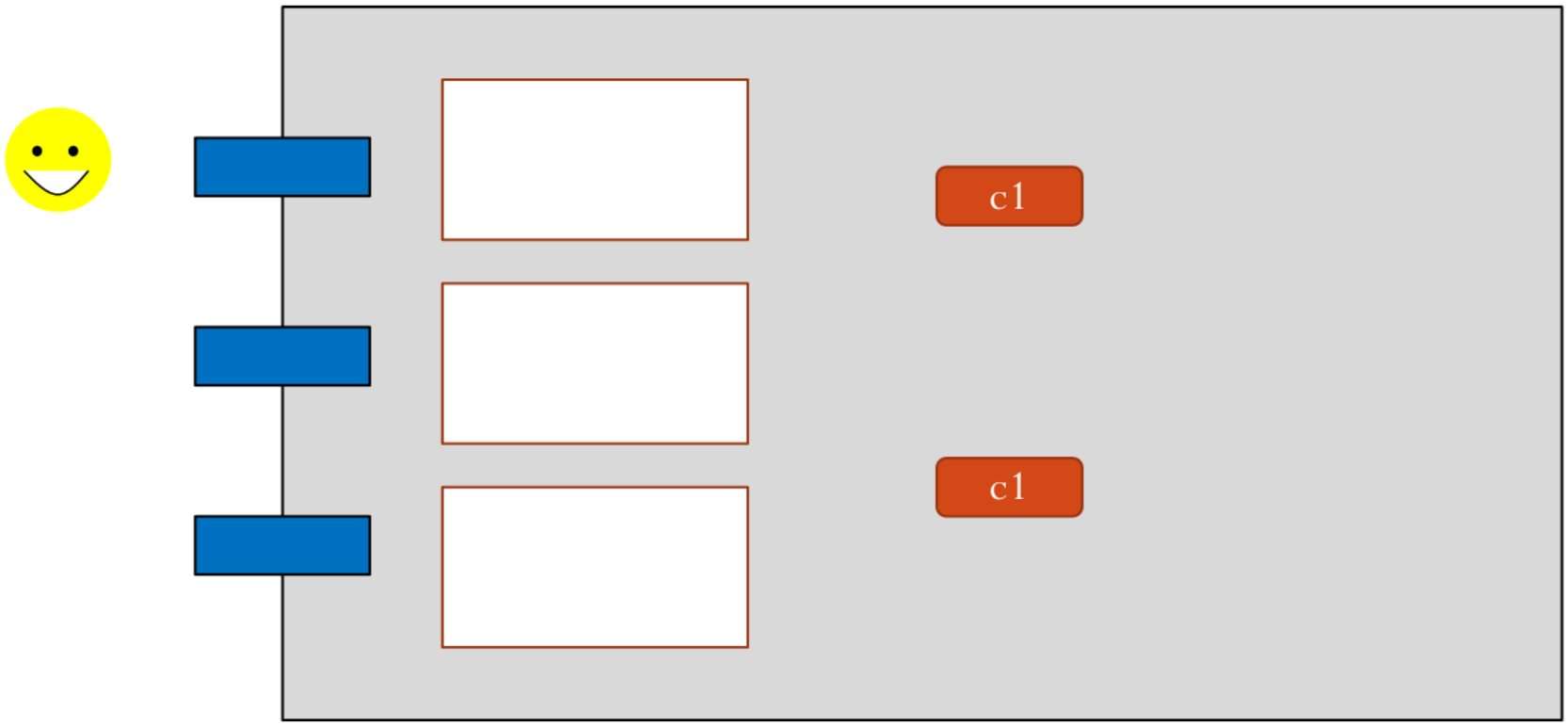
Condition variables

- Están ligadas al monitor.
- Poseen dos operaciones:
  - `wait()`
  - `notify()`
- Al igual que los semáforos, los monitores tienen asociadas una cola de procesos bloqueados.

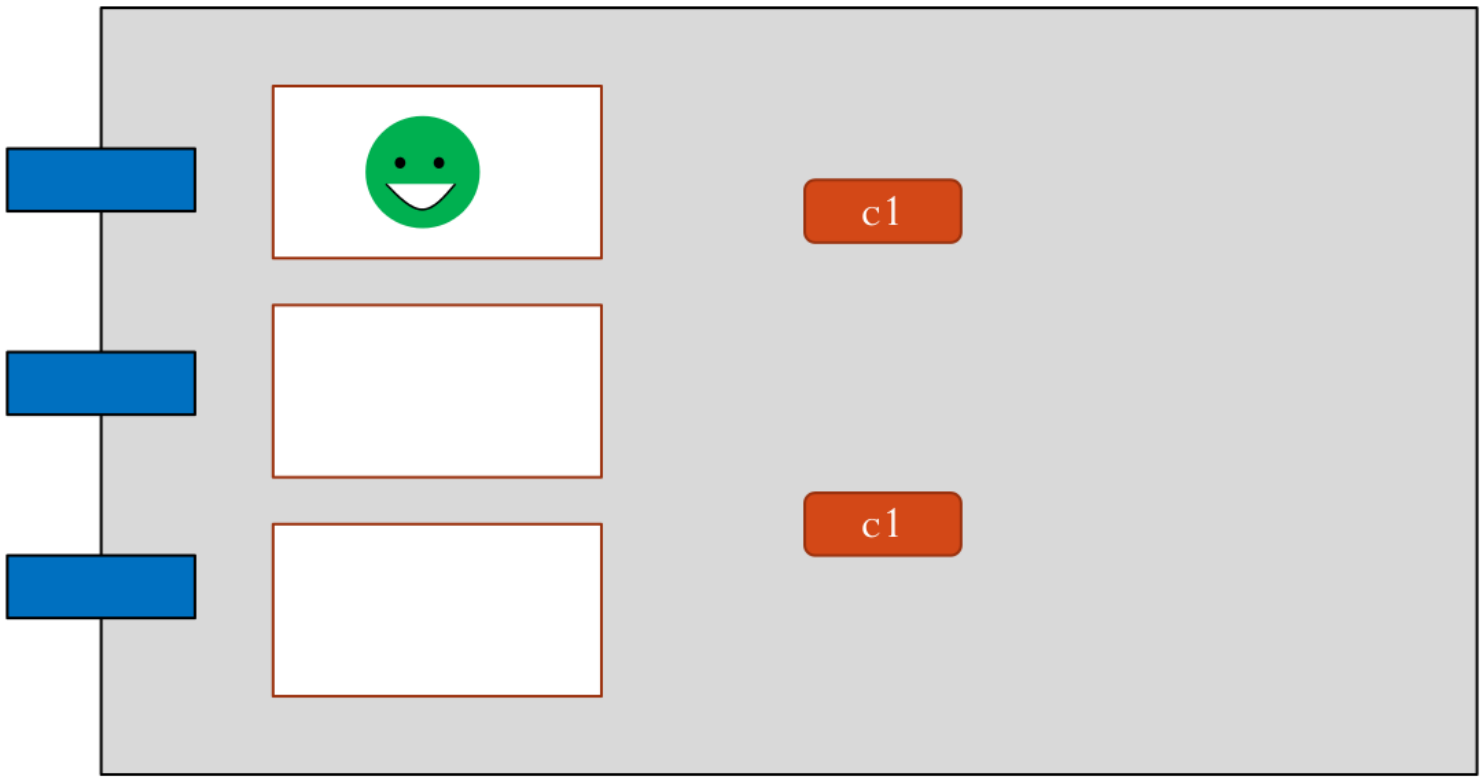
Gráficamente



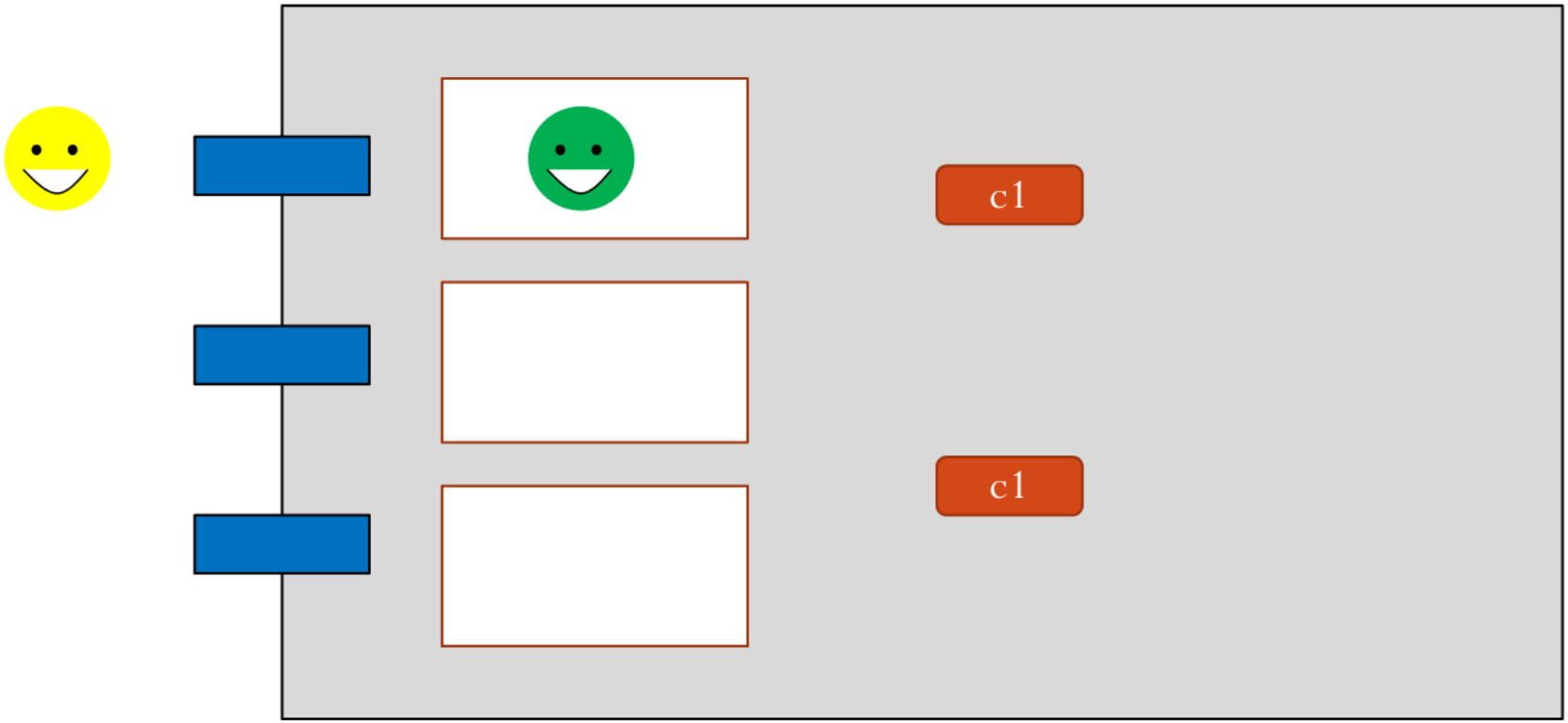
Comportamiento típico



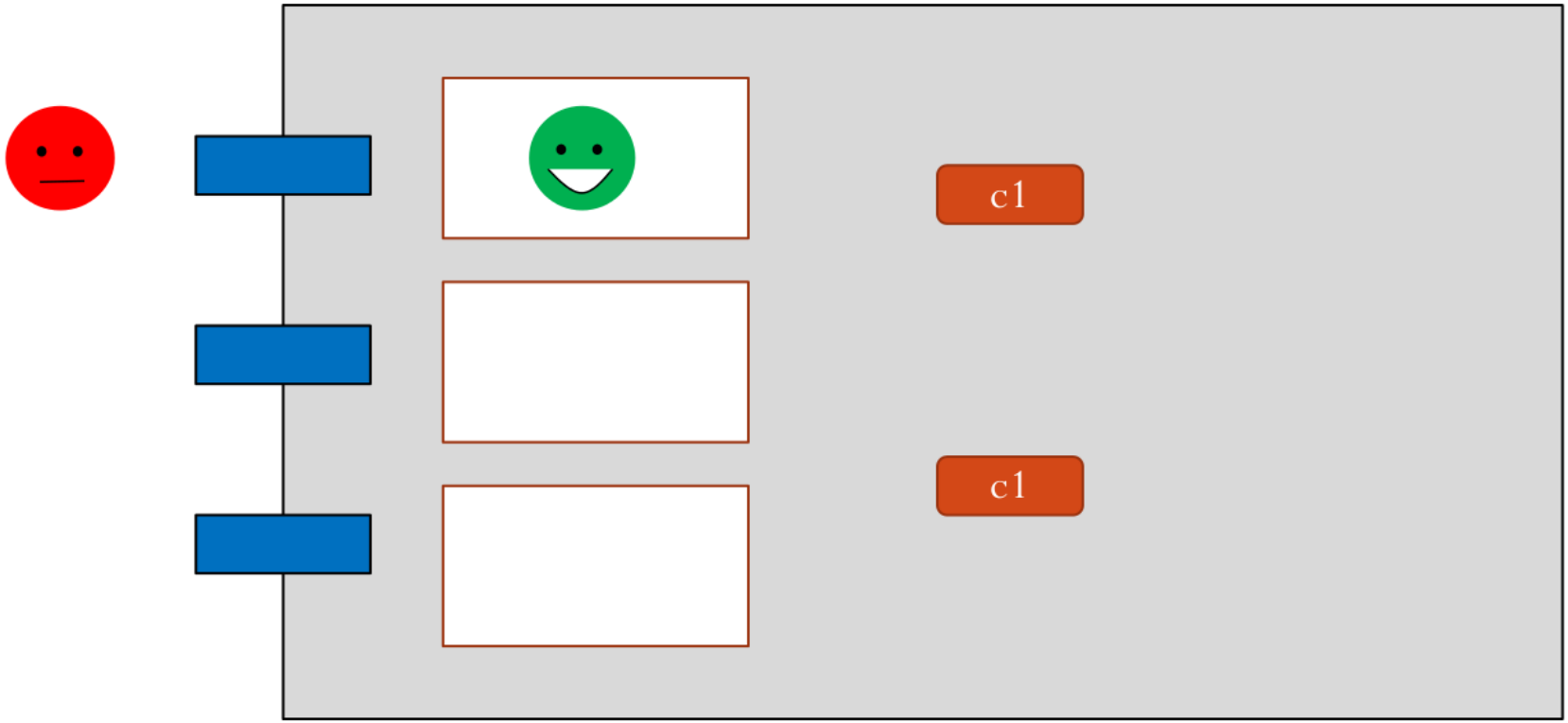
Comportamiento típico



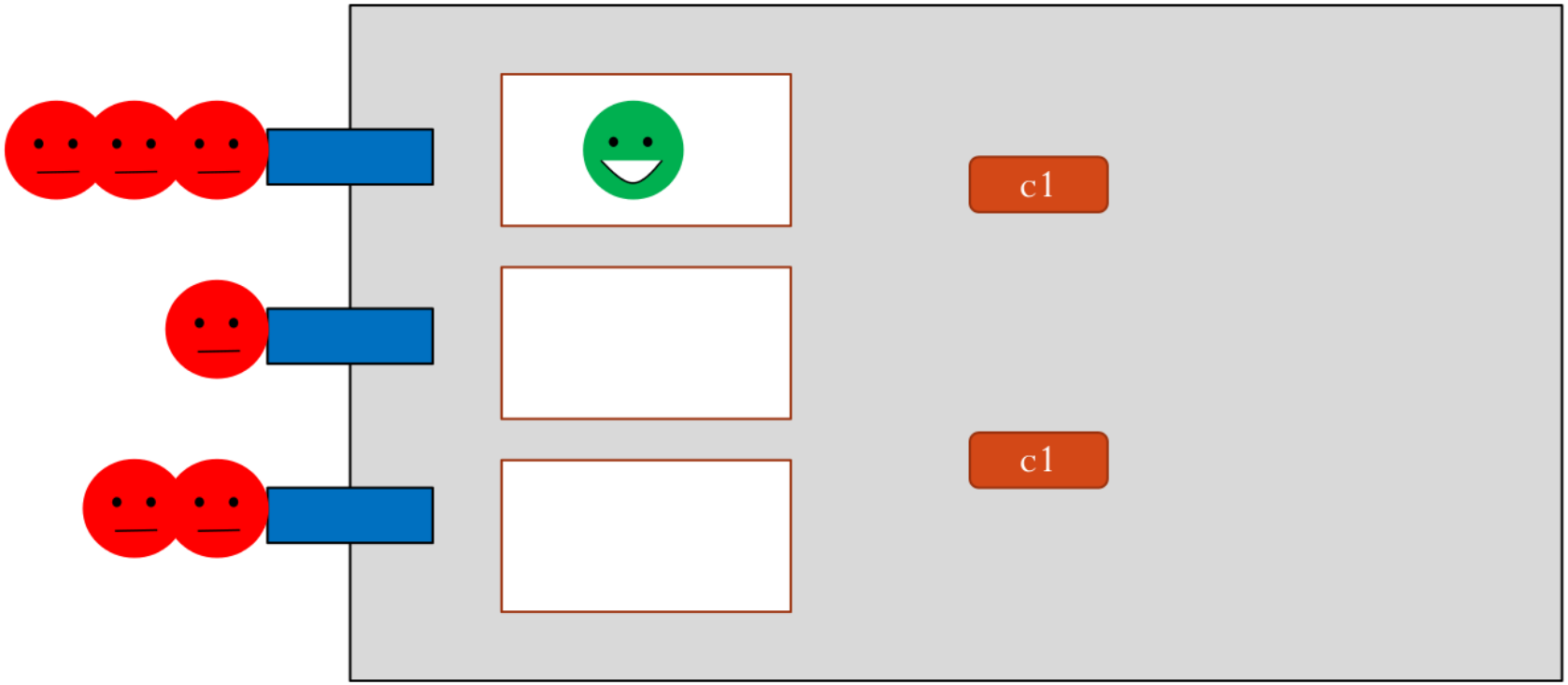
Comportamiento típico



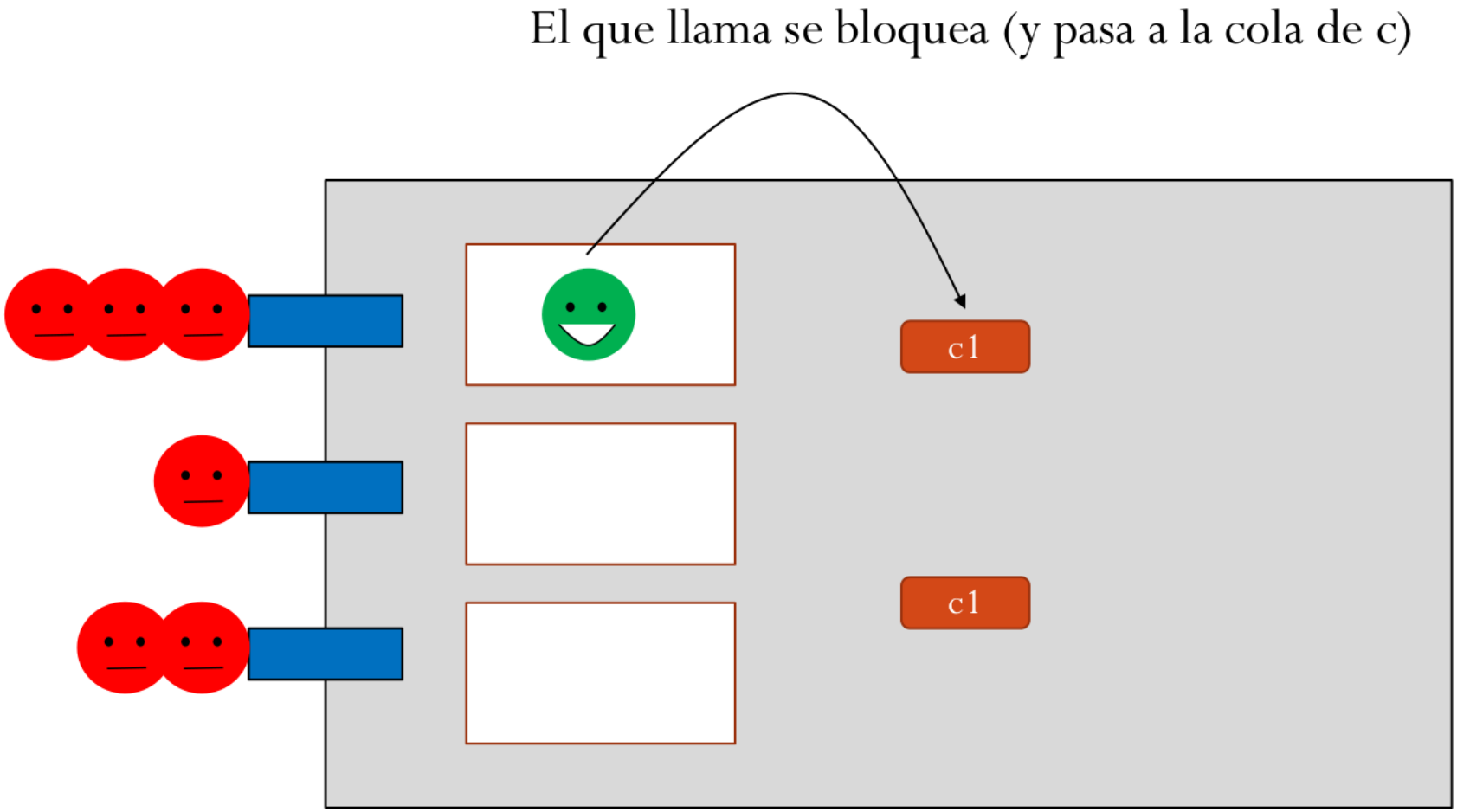
Comportamiento típico



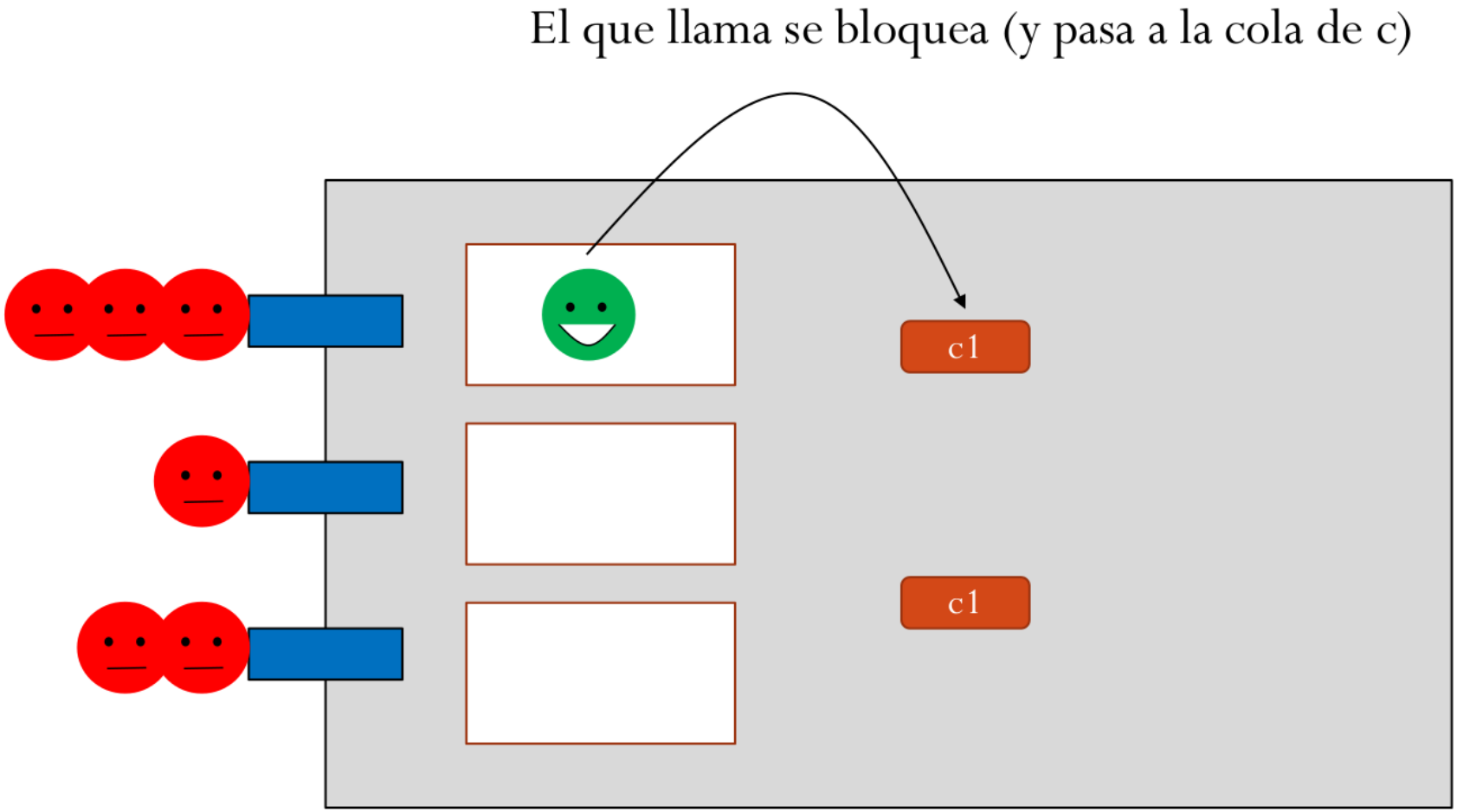
Comportamiento típico



Wait

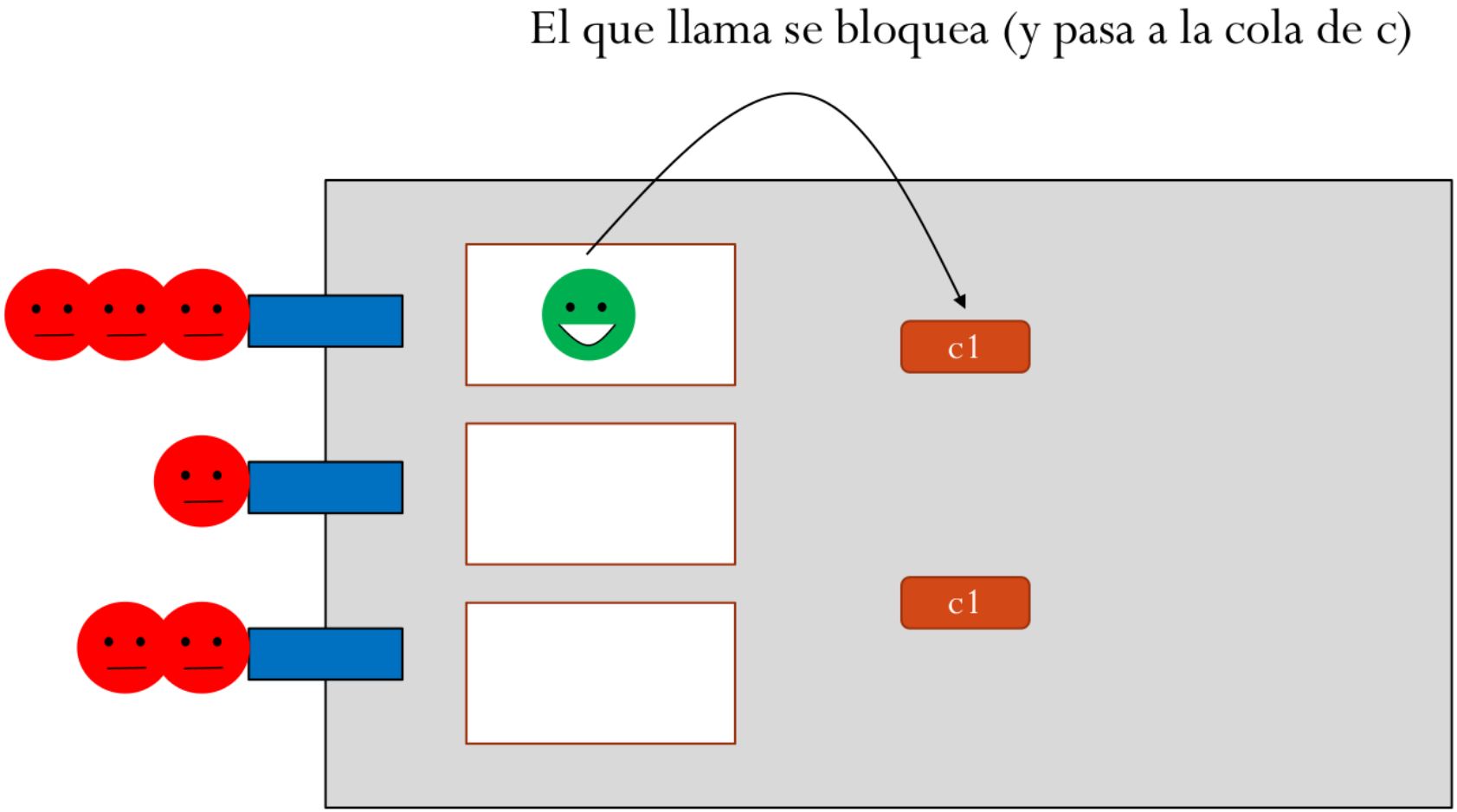


Wait



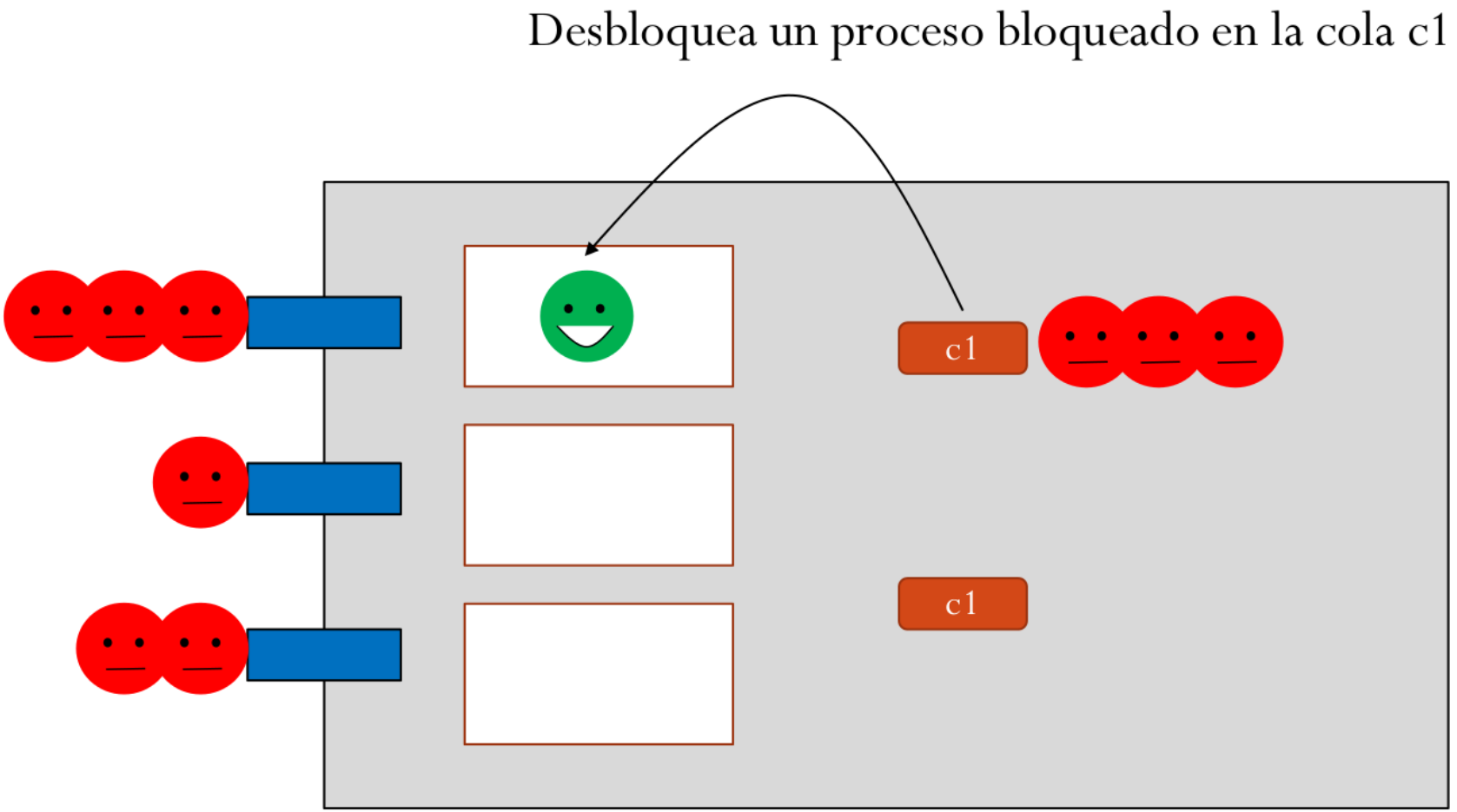
- Bloquea al proceso en ejecución y lo asocia a la variable

Wait

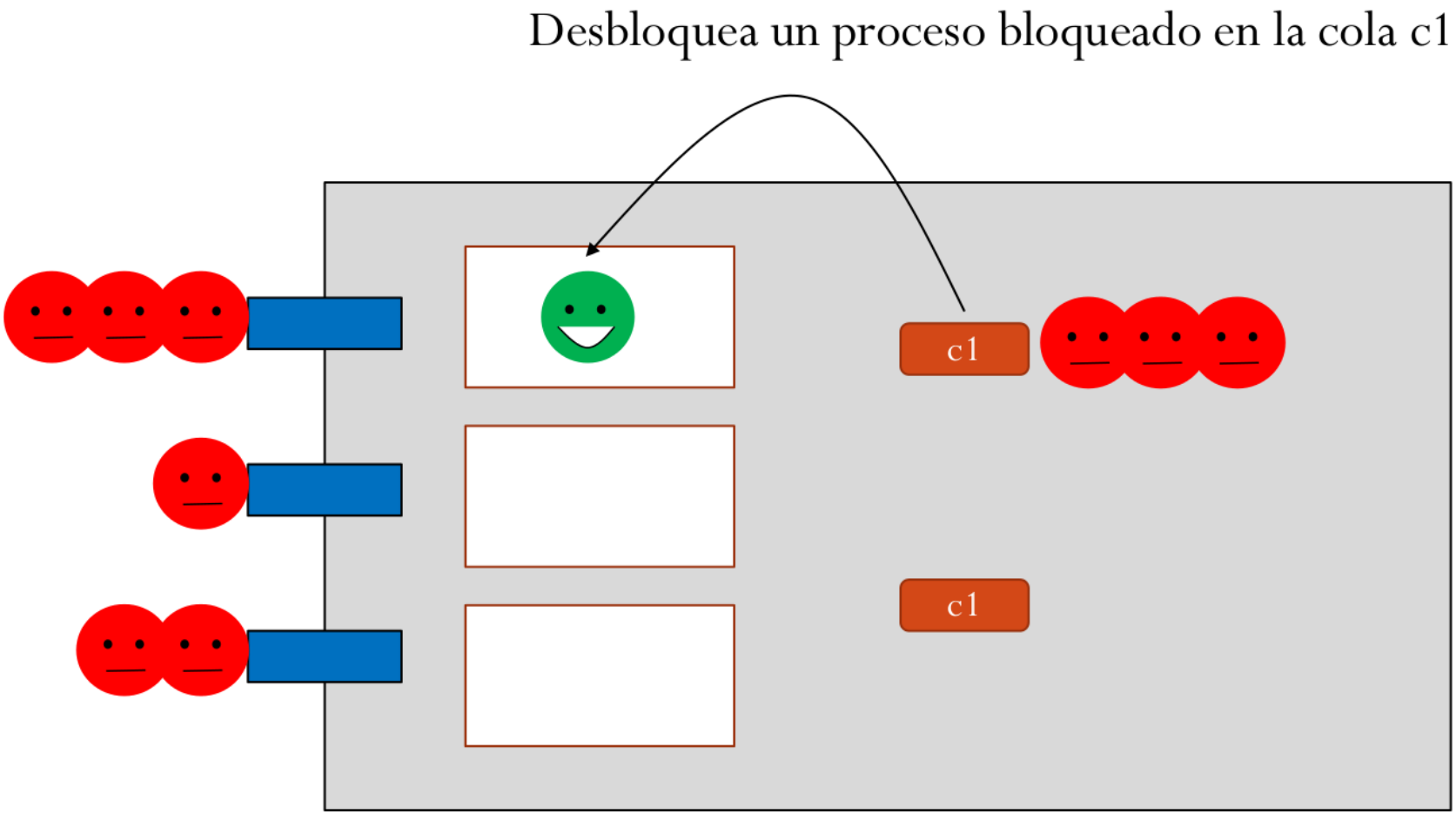


- Bloquea al proceso en ejecución y lo asocia a la variable
- Al bloquearse libera el *lock* permitiendo la entrada de otro

Notify

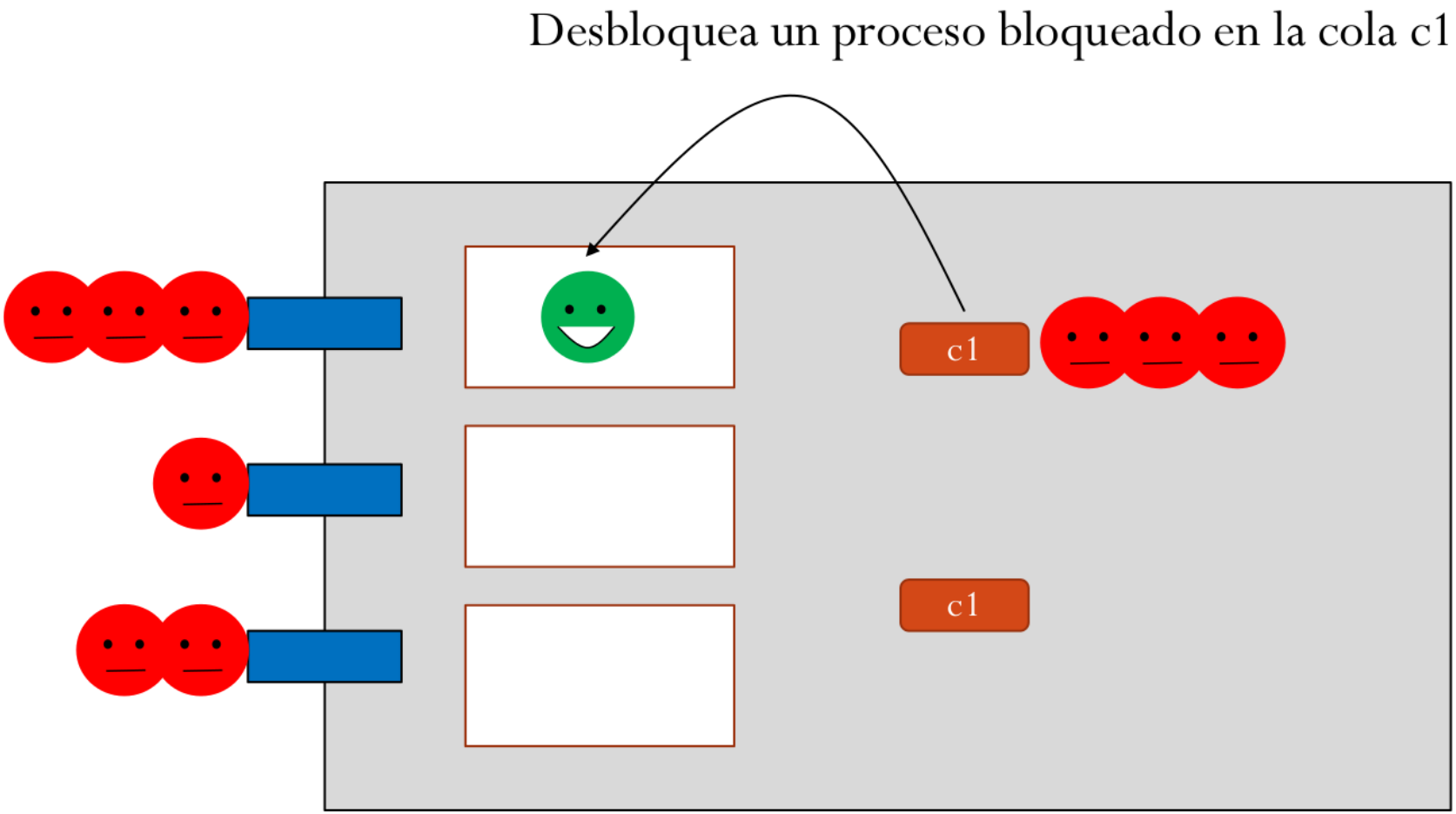


Notify



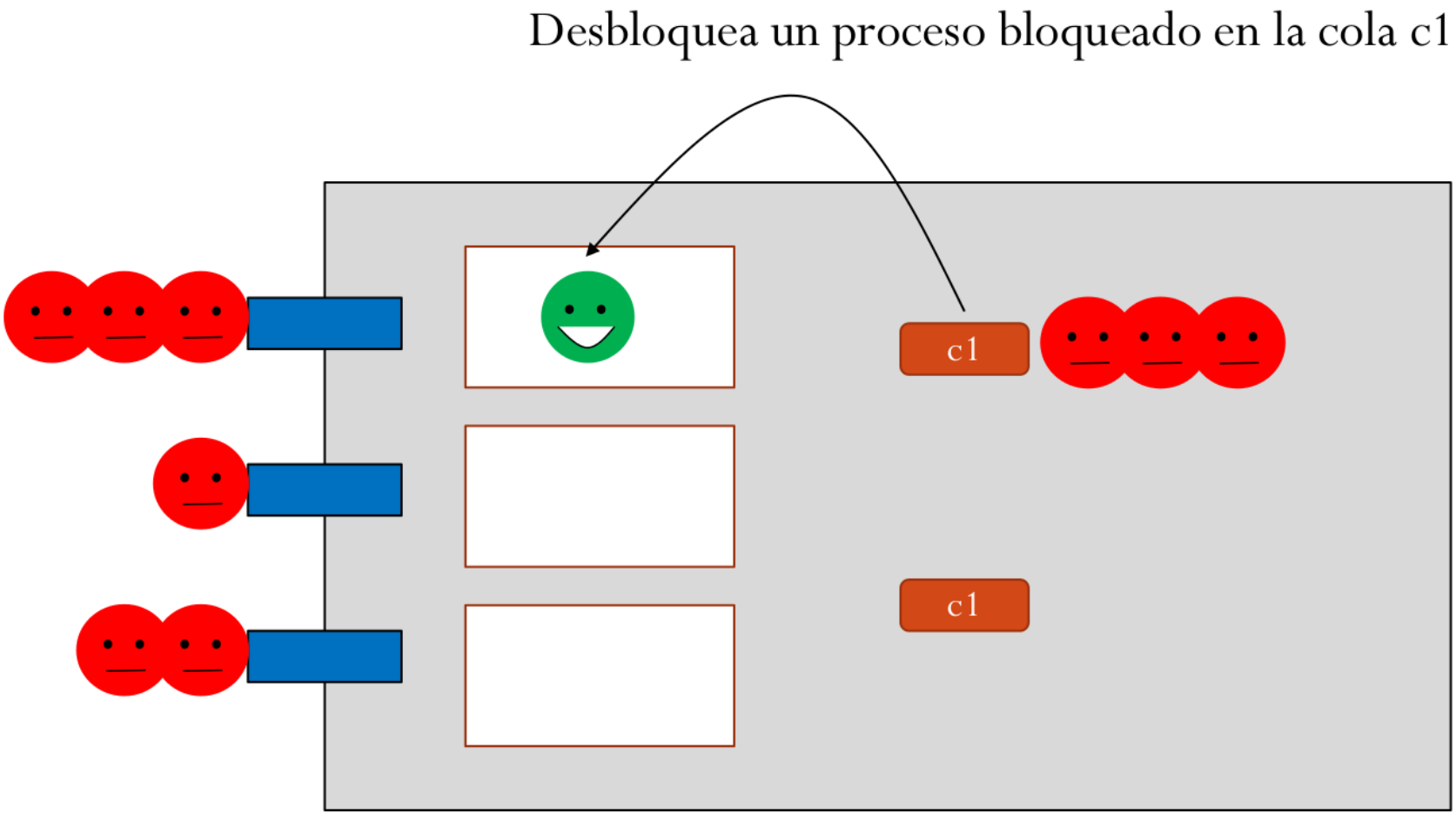
- Desbloquea al primer proceso de la variable de condición.

Notify



- Desbloquea al primer proceso de la variable de condición.
- ¿Qué sucede con el *lock*?

Notify



- Desbloquea al primer proceso de la variable de condición.
- ¿Qué sucede con el *lock*?
- ¿En qué se diferencia de un `release` de un semáforo?

Notify y salida de ejecución

Notify y salida de ejecución

- Cuando un proceso es desbloqueado continua su ejecución desde la instrucción siguiente a la llamada del `wait` que lo bloqueó.

Notify y salida de ejecución

- Cuando un proceso es desbloqueado continua su ejecución desde la instrucción siguiente a la llamada del `wait` que lo bloqueó.
- ¿Qué sucede con el *lock*?

Notify y salida de ejecución

- Cuando un proceso es desbloqueado continua su ejecución desde la instrucción siguiente a la llamada del `wait` que lo bloqueó.
- ¿Qué sucede con el *lock*?
  - Para continuar la ejecución el proceso desbloqueado debe adquirir el *lock*.

Ejemplo: Buffer

Ejemplo: Buffer

```
monitor Buffer {
  private condition hayEspacio; // esperar a que haya espacio
  private condition hayDato;    // esperar a que haya dato

  private Object dato = null;   // el dato compartido

  public Object read() {
    while (dato == null)
      hayDato.wait();
    aux = dato;
    dato = null;
    hayEspacio.notify();
    return aux;
  }

  public void write(Object o) {
    while (dato != null)
      hayEspacio.wait();
    dato = o;
    hayDato.notify();
  }

}
```

Ejercicio: Buffer de dimensión N

Monitor que define a un semáforo

Monitor que define a un semáforo

```
monitor Semaphore {
  private condition noCero;
  private int permisos;

  public Semaphore(int n) {
    this.permisos = n;
  }

  public void acquire() {
    while (permisos == 0)
      noCero.wait();
    permisos--;
  }

  public void release() {
    permisos++;
    noCero.notify();
  }

}
```

- ¿Tiene algún problema esta solución?

Monitor que define un semáforo II

```
public void acquire() {
  waitingThreads.enqueue(Thread.currentThread());
  while (permissions == 0) {
    wait();
    if (passedPermissions > 0 &&
        waitingThreads.contains(Thread.currentThread()) {
      passedPermissions--;
      waitingThreads.remove(Thread.currentThread();
      return;
    }
  }
  permissions--;
}

public void release() {
  if (waitingThreads.isEmpty()) {
    permissions++;
  } else {
    passedPermissions++;
    notify();
  }
}
```

Monitores en Java

Monitores en Java

- Toda clase tiene un lock y una única variable de condición.
  - Ventaja: Queda bien encapsulado (no se puede manipular el lock ni la variable de condición)

Monitores en Java

- Toda clase tiene un lock y una única variable de condición.
  - Ventaja: Queda bien encapsulado (no se puede manipular el lock ni la variable de condición)
  - Desventaja: Usar más de una variable de condición puede mejorar la eficiencia
- Los métodos `wait`, `notify` y `notifyAll` pertenecen a la interfaz de la clase `Object`

Monitores en Java

- Toda clase tiene un lock y una única variable de condición.
  - Ventaja: Queda bien encapsulado (no se puede manipular el lock ni la variable de condición)
  - Desventaja: Usar más de una variable de condición puede mejorar la eficiencia
- Los métodos `wait`, `notify` y `notifyAll` pertenecen a la interfaz de la clase `Object`
- Es necesario usar el keyword `synchronized` en cada método del monitor
  - Garantiza exclusión mutua
  - Permite invocar las operaciones sobre la variable de condición

Buffer de dimensión N en Java

```
class Buffer {
  private Object[] data = new Object[N+1];
  private int begin = 0, end = 0;

  public synchronized void push(Object o) {
    while (isFull()) wait();
    data[begin] = o;
    begin = next(begin);
    notifyAll();
  }

  public synchronized Object pop() {
    while (isEmpty()) wait();
    Object result = data[end];
    end = next(end);
    notifyAll();
    return result;
  }

  private boolean isEmpty() { return begin == end; }
  private boolean isFull()  { return next(begin) == end; }
  private int next(int i)   { return (i+1)%(N+1); }

}
```

IllegalMonitorStateException

- Sólo se pueden invocar los métodos wait, notify y notifyAll desde métodos synchronized.
- En caso contrario se emite una excepción.

```
public void m1() {
  this.wait(); // IllegalMonitorStateException
}
```

```
public void m1() {
  this.notify(); // IllegalMonitorStateException
}
```

Threads en Java (Productor)

```
class Productor extends Thread {

  private final Buffer buffer;

  public Productor(Buffer buffer) {
    this.buffer = buffer;
  }

  public void run() {
    int i = 0;
    while (true) {
      buffer.write(i);
      i++;
    }
  }

}
```

Threads en Java (Consumidor)

```
class Consumidor implements Runnable {

  private final Buffer buffer;

  public Consumidor(Buffer buffer) {
    this.buffer = buffer;
  }

  public void run() {
    while (true) {
      Object o = buffer.read();
      System.out.println("Leido " + o.toString());
    }
  }

}
```

Threads en Java (thread principal)

```
public static void main(String[] args) {
  Buffer buffer = new Buffer();
  Productor p   = new Productor(buffer);
  Consumidor c = new Consumidor(buffer);
  Thread ct     = new Thread(c);

  p.start();
  ct.start();
}
```

El método `start` inicia un nuevo thread que ejcutará de forma concurrente con el original. El nuevo thread ejecutará internamente el método `run`.

InterruptedException

- El método wait puede arrojar una excepción.
- Esto ocurre cuando se interrumpe al thread en espera usando el método interrupt.
- Aunque no lo usemos es una buena práctica considerar eventuales interrupciones.

Lectores Escritores con Monitores

```
class Database {

  synchronized void beginWrite();

  synchronized void endWrite();

  synchronized void beginRead();

  synchronized void endRead();

}
```

Lectores Escritores Solución

```
class Database {

  private int writers = 0;
  private int readers = 0;

  private boolean canRead() {
    return writers == 0;
  }

  private boolean canWrite() {
    return writers == 0 && readers == 0;
  }

  ...

}
```

Lectores Escritores Solución

```
syncrhonized void beginRead() {
  while (!canRead()) {
    try { wait(); }
    catch (InterruptedException e) { return; }
  }
  readers++;
}

syncrhonized void endRead() {
  readers--;
  if (readers == 0)
    notify();
}
```

Lectores Escritores Solución

```
syncrhonized void beginWrite() {
  while (!canWrite()) {
    try { wait(); }
    catch (InterruptedException e) { return; }
  }
  writers = 1;
}

syncrhonized void endWrite() {
  writers = 0;
  notifyAll();
}
```

Lectores Escritores (Prioridad Escritores)

```
private int waitingWriters = 0;

private boolean canRead() {
  return writers == 0 && waitingWriters == 0;
}

syncrhonized void beginWrite() {
  while (!canWrite()) {
    waitingWriters++;
    try { wait(); }
    catch (InterruptedException e) { return; }
    finally { waitingWriters--; }
  }
  writers = 1;
}

syncrhonized void endWrite() {
  writers = 0;
  notifyAll();
}
```