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.
- Un mecanismo de espera llamado *condition variable*, utilizadas para programar sincronización condicional.

Ejemplo: Contador

```
monitor Contador {

  private int contador = 0;

  public void incrementar() {
    contador++;
  }

  public void decrementar() {
    contador--;
  }

}
```

Ejemplo: Contador

```
monitor Contador {

  private int contador = 0;

  public void incrementar() {
    contador++;
  }

  public void decrementar() {
    contador--;
  }

}
```

- No puede haber interferencia (e.g., pérdida de sumas/restas) porque las operaciones se ejecutan en exclusión mutua.

Condition variables

Condition variables

- Están ligadas al monitor.

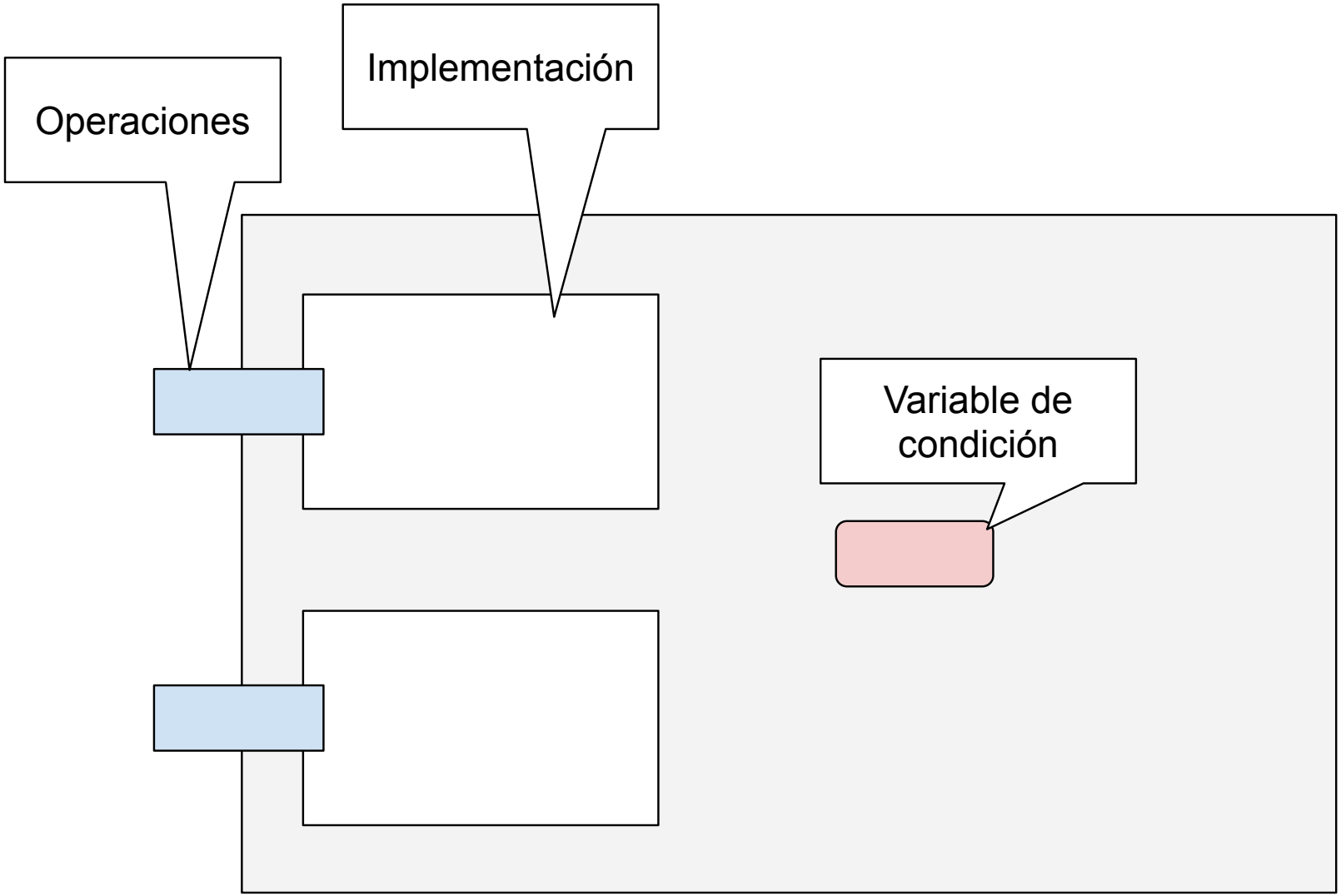
Condition variables

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

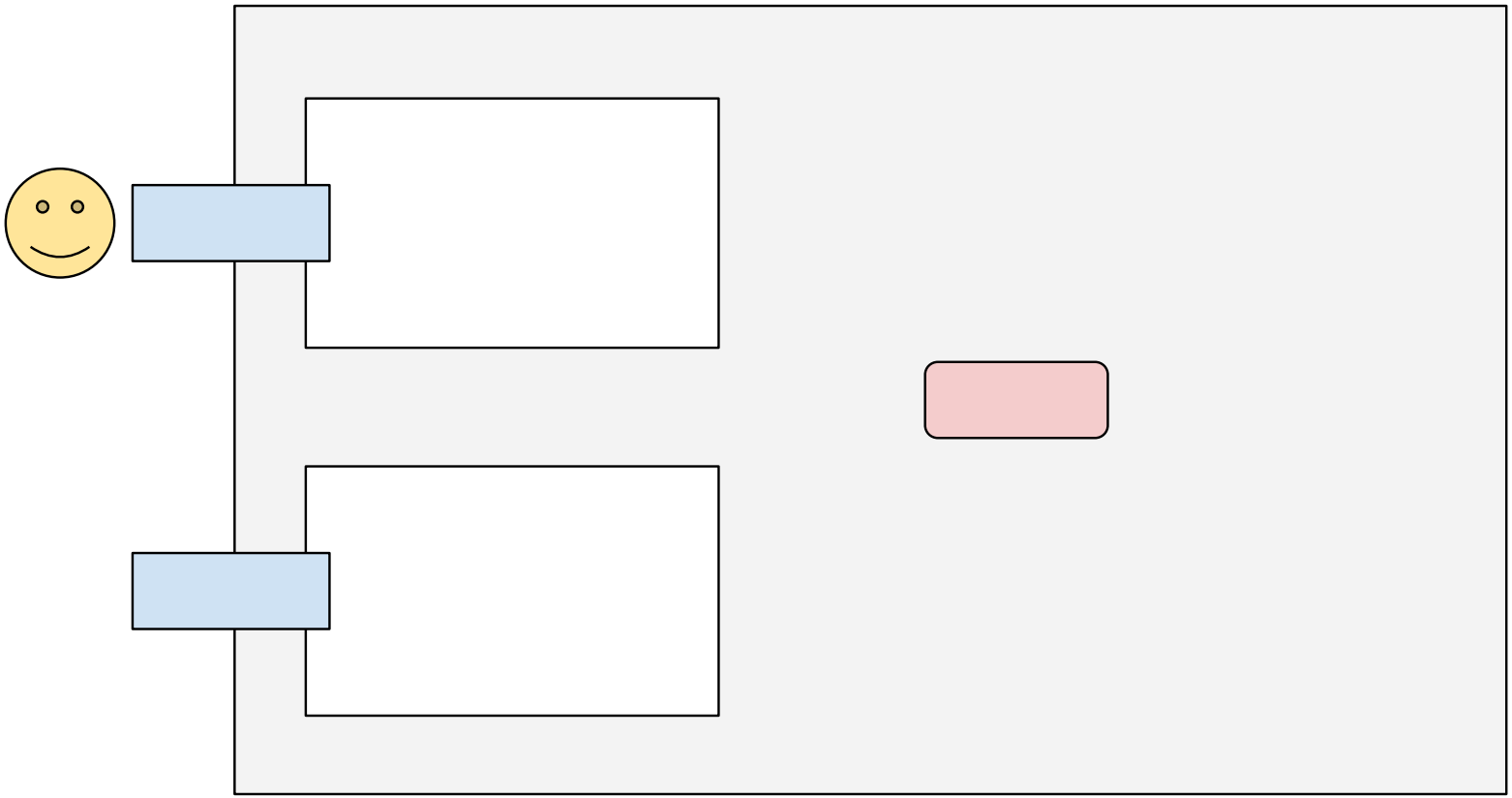
Condition variables

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

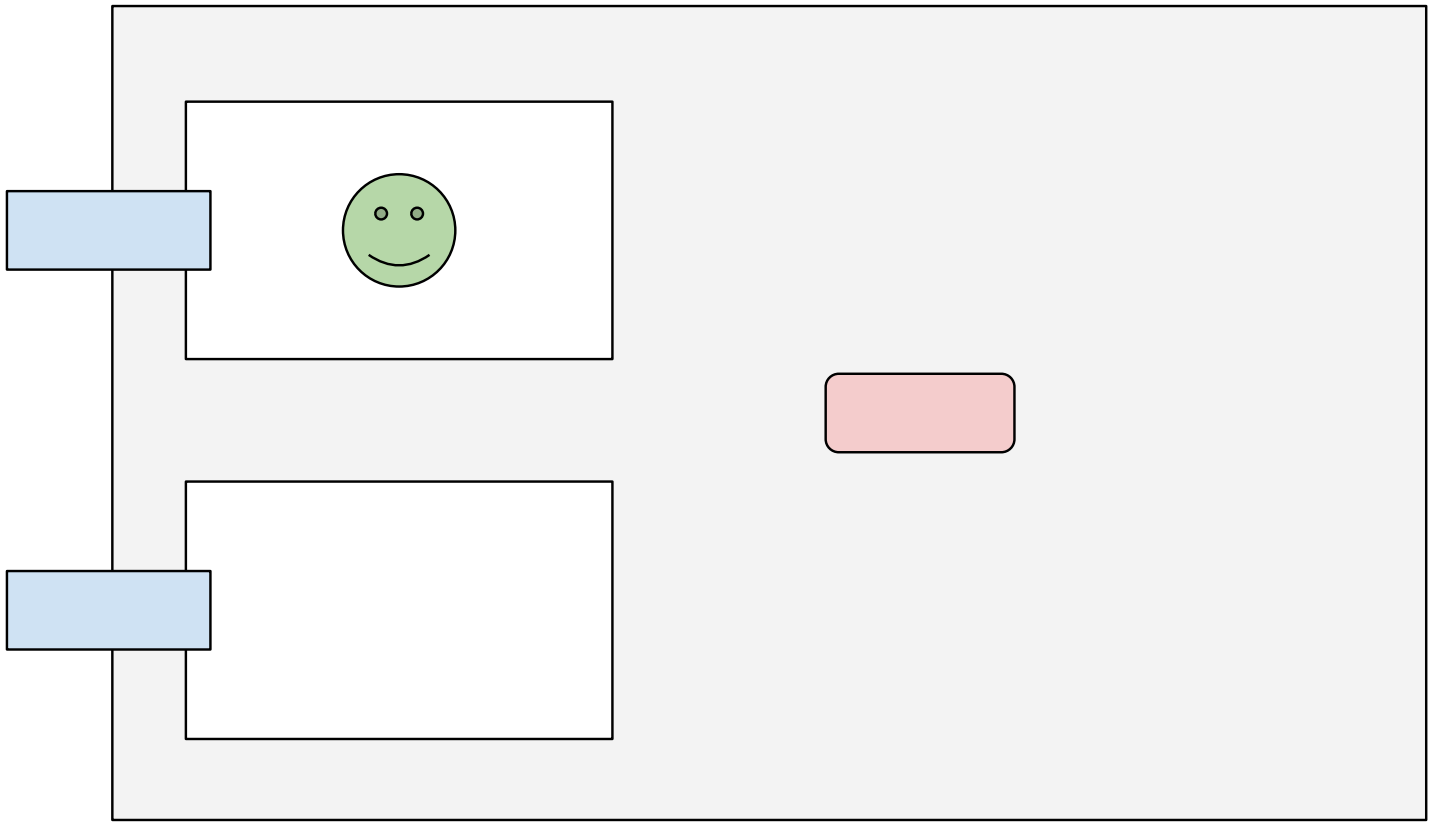
Comportamiento de un monitor (gráficamente)



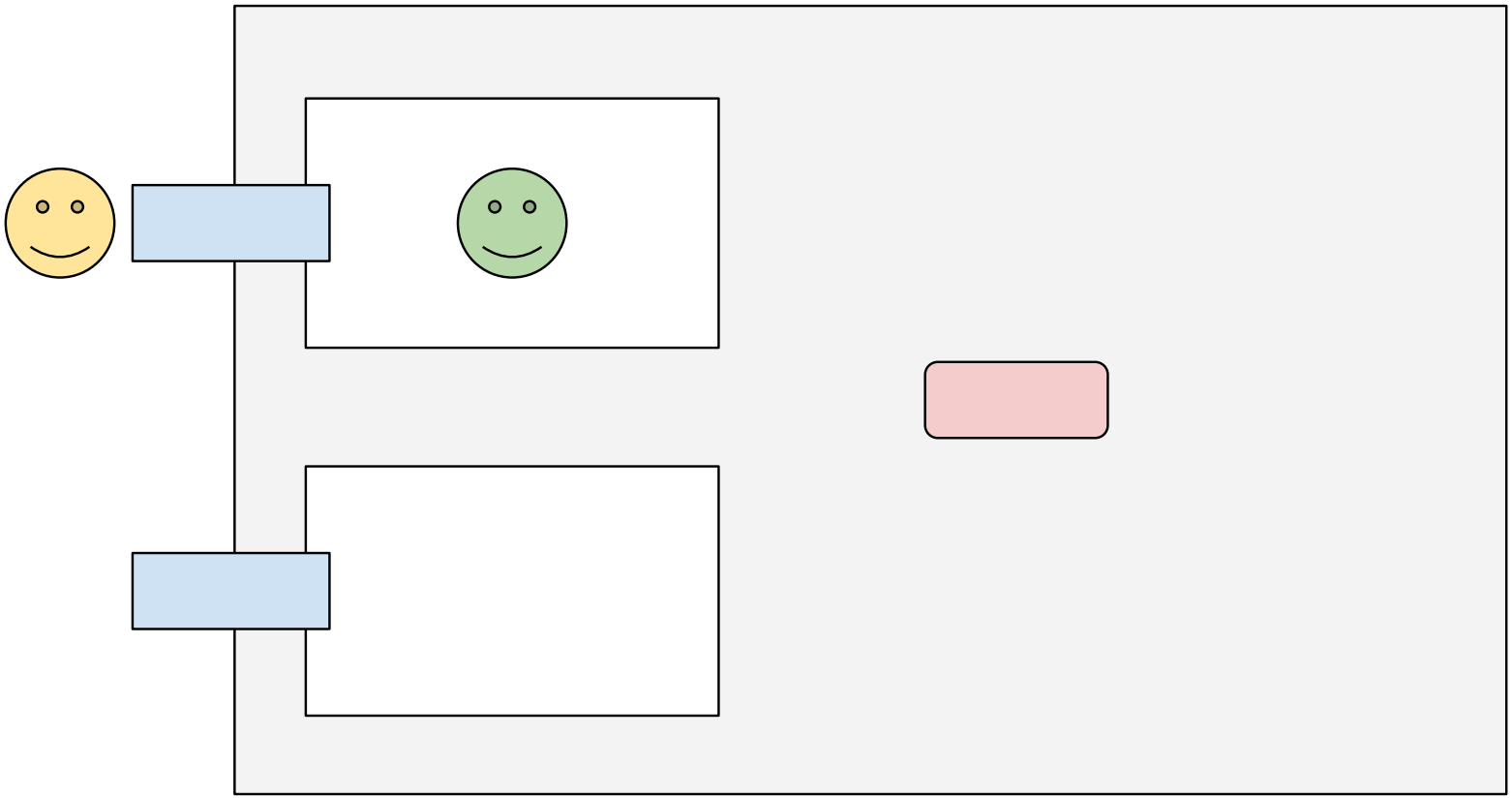
Comportamiento de un monitor (gráficamente)



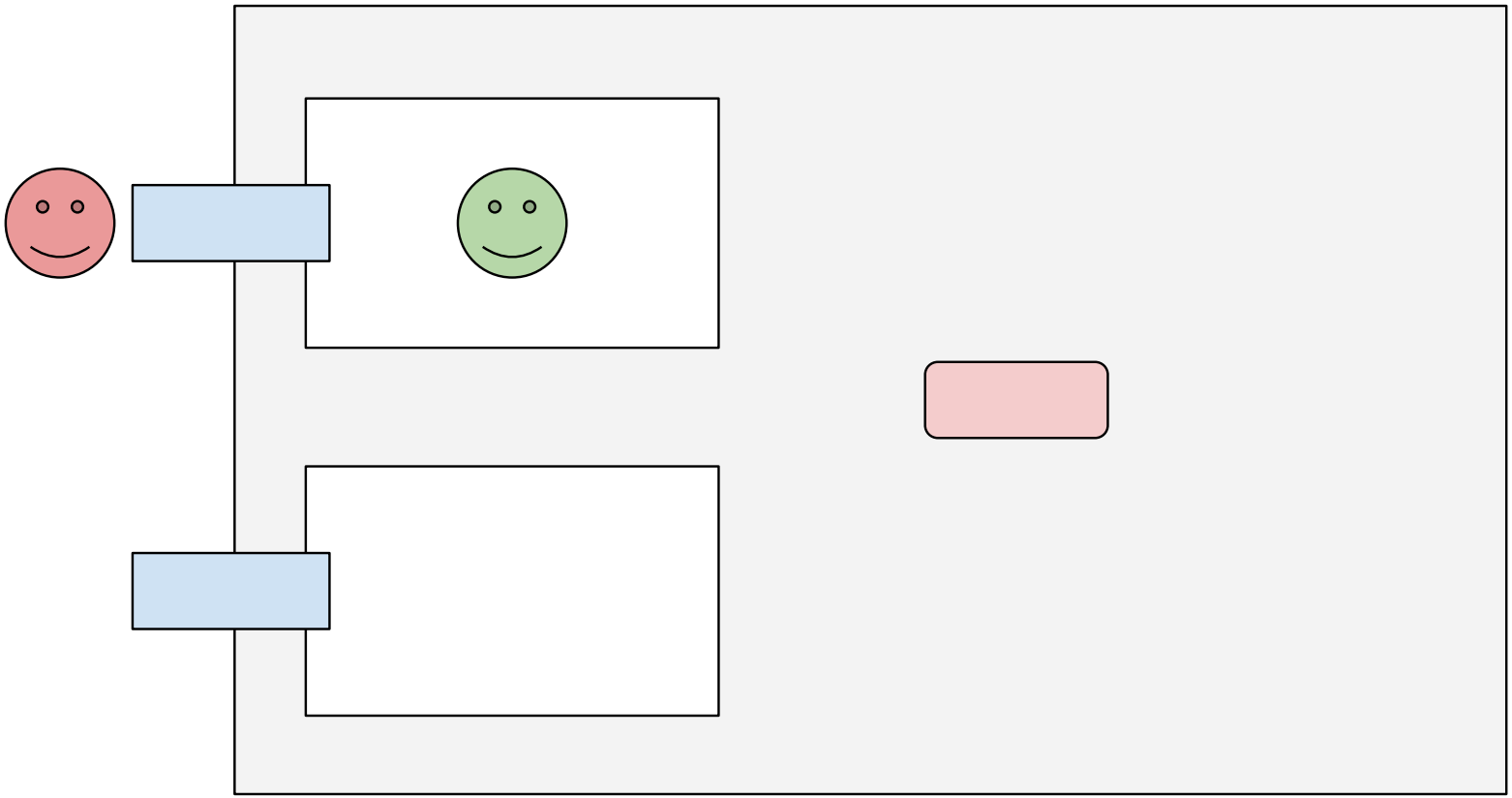
Comportamiento de un monitor (gráficamente)



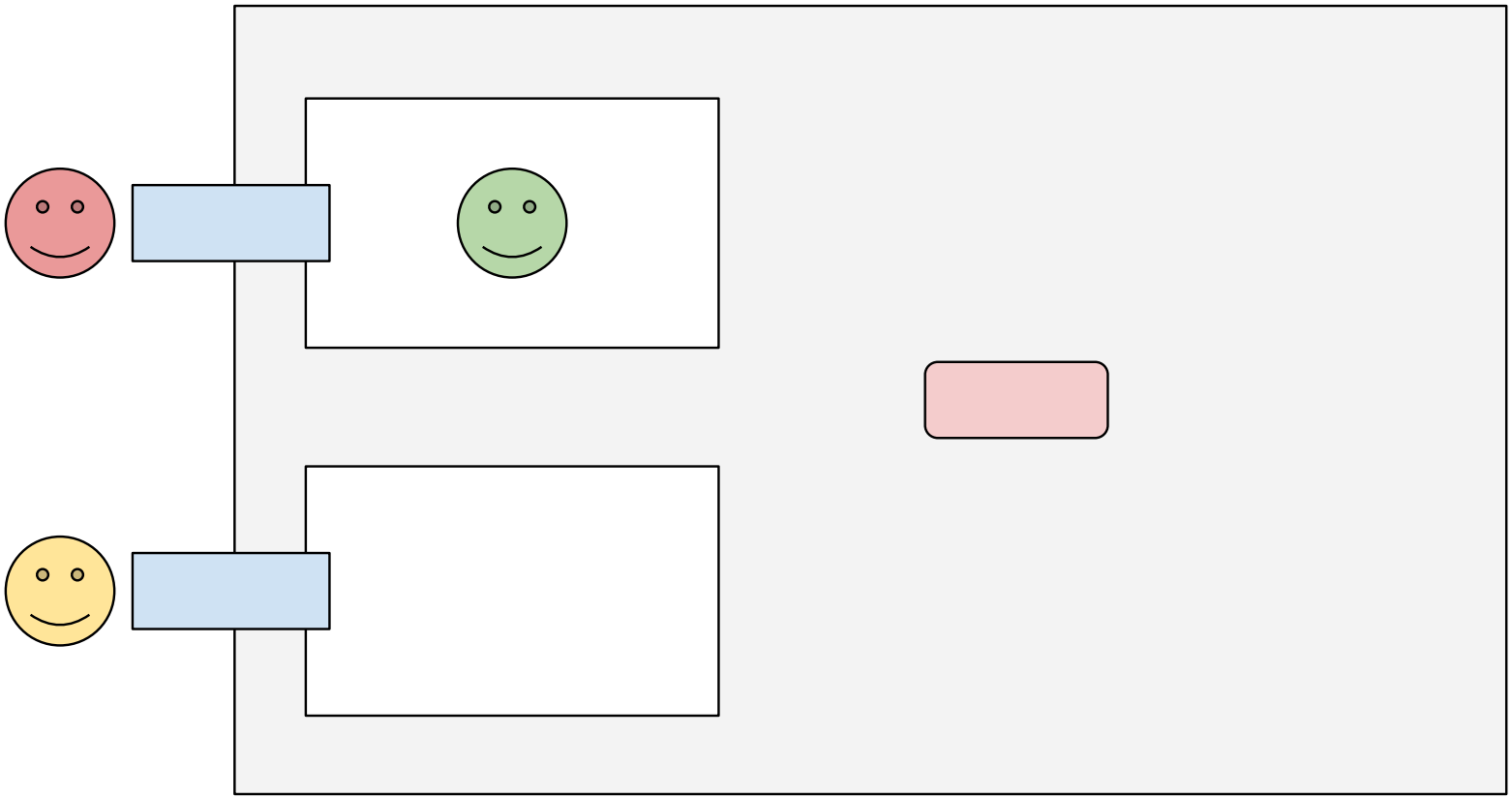
Comportamiento de un monitor (gráficamente)



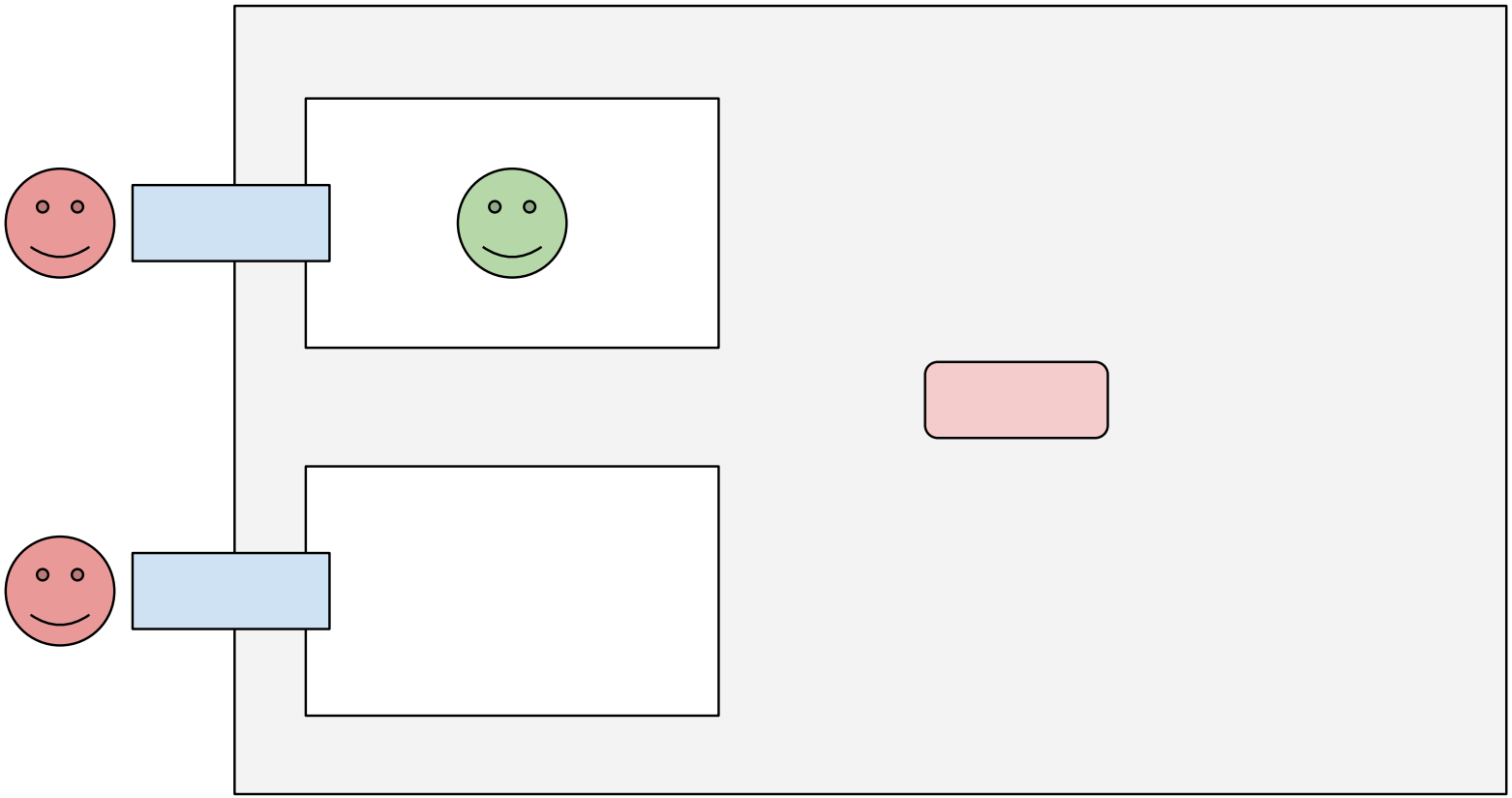
Comportamiento de un monitor (gráficamente)



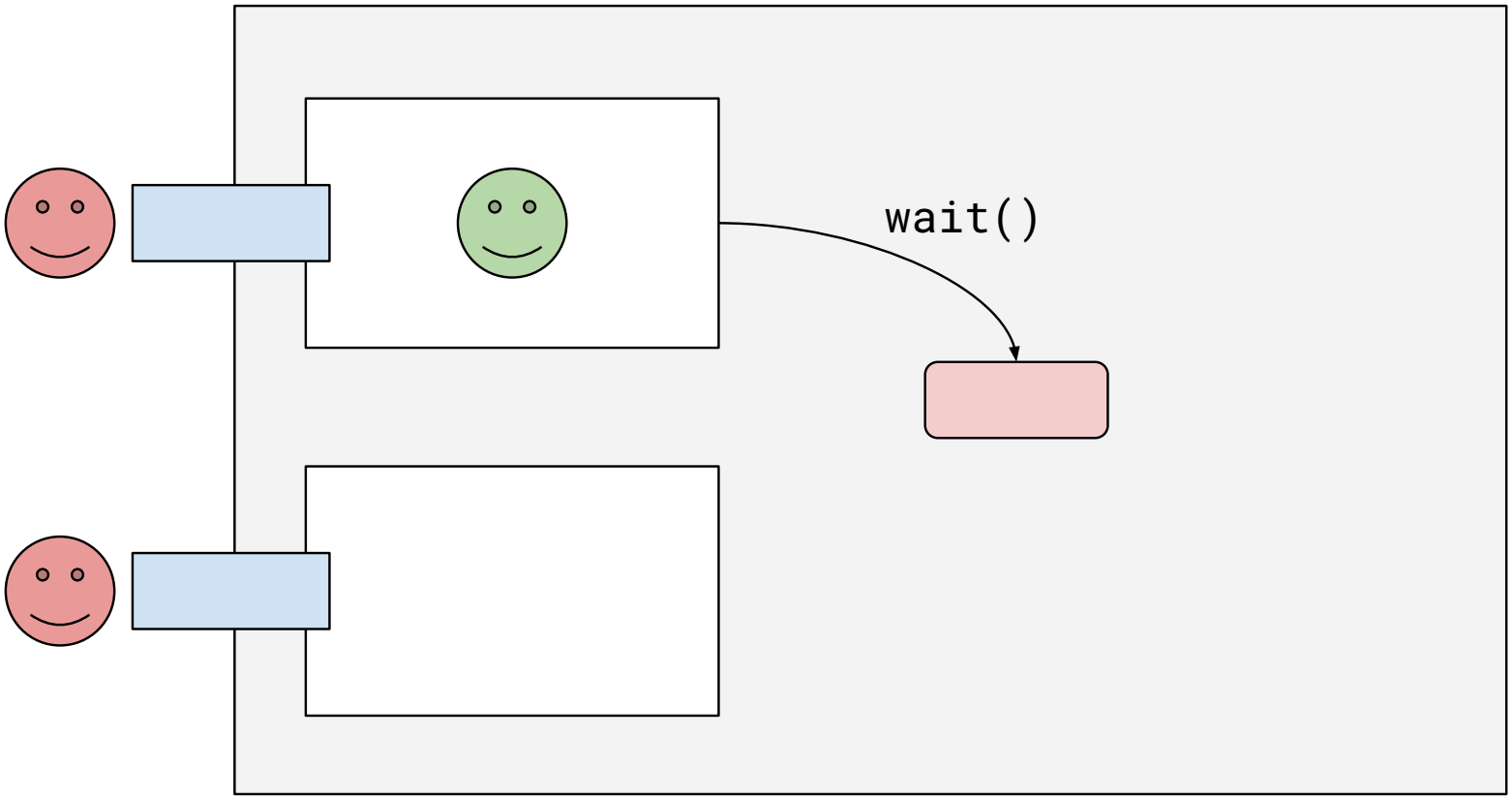
Comportamiento de un monitor (gráficamente)



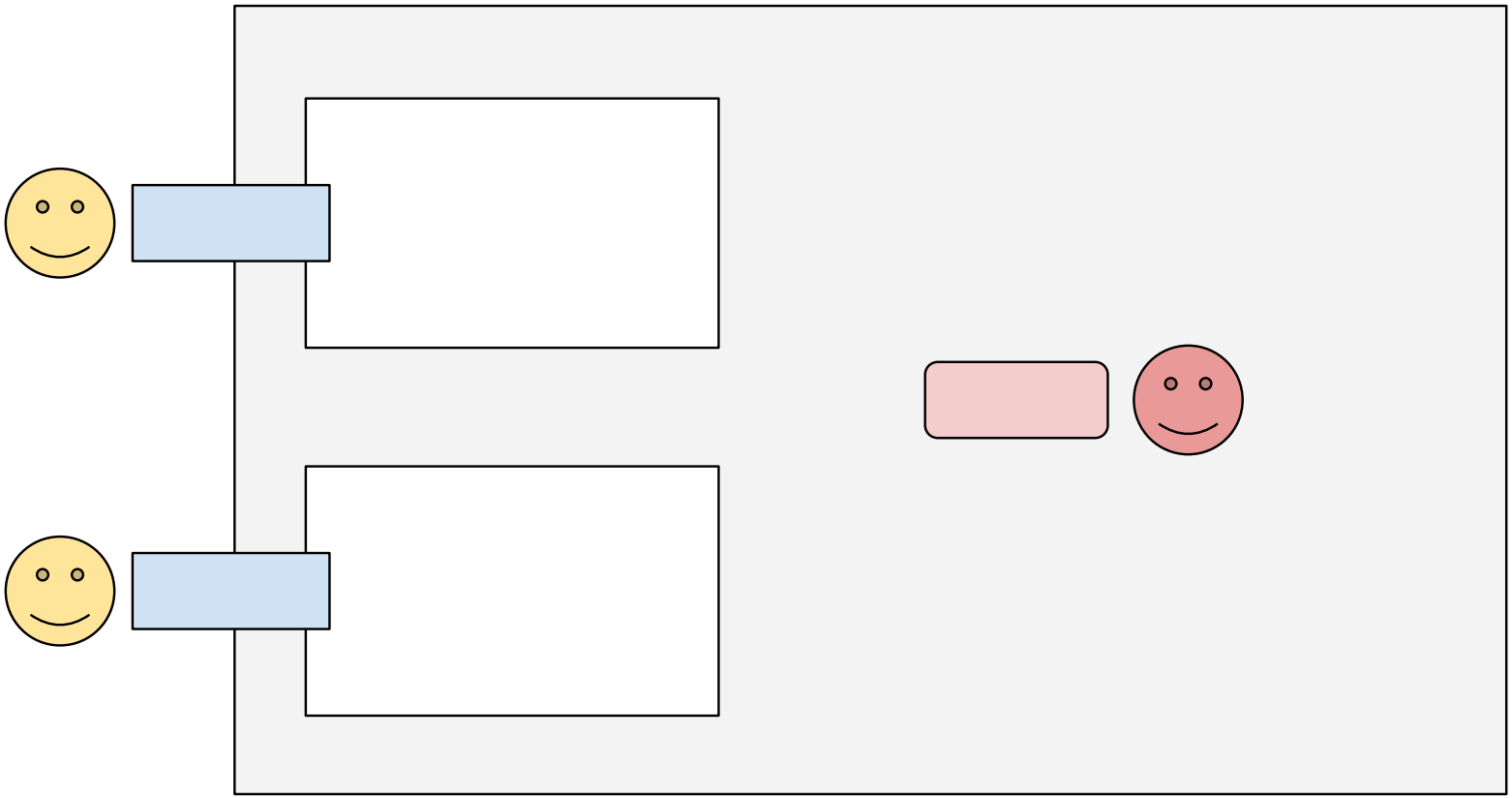
Comportamiento de un monitor (gráficamente)



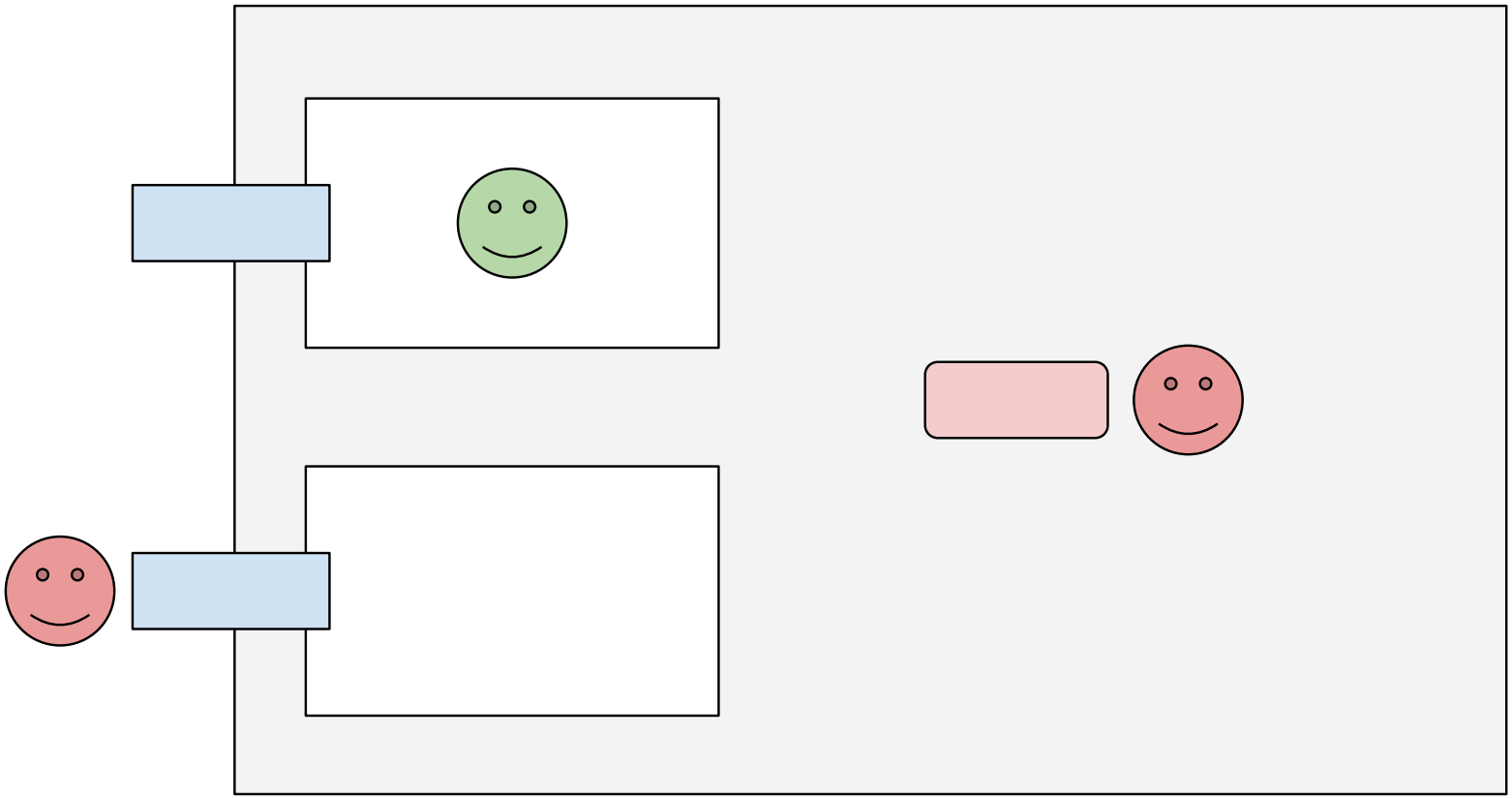
Comportamiento de un monitor (gráficamente)



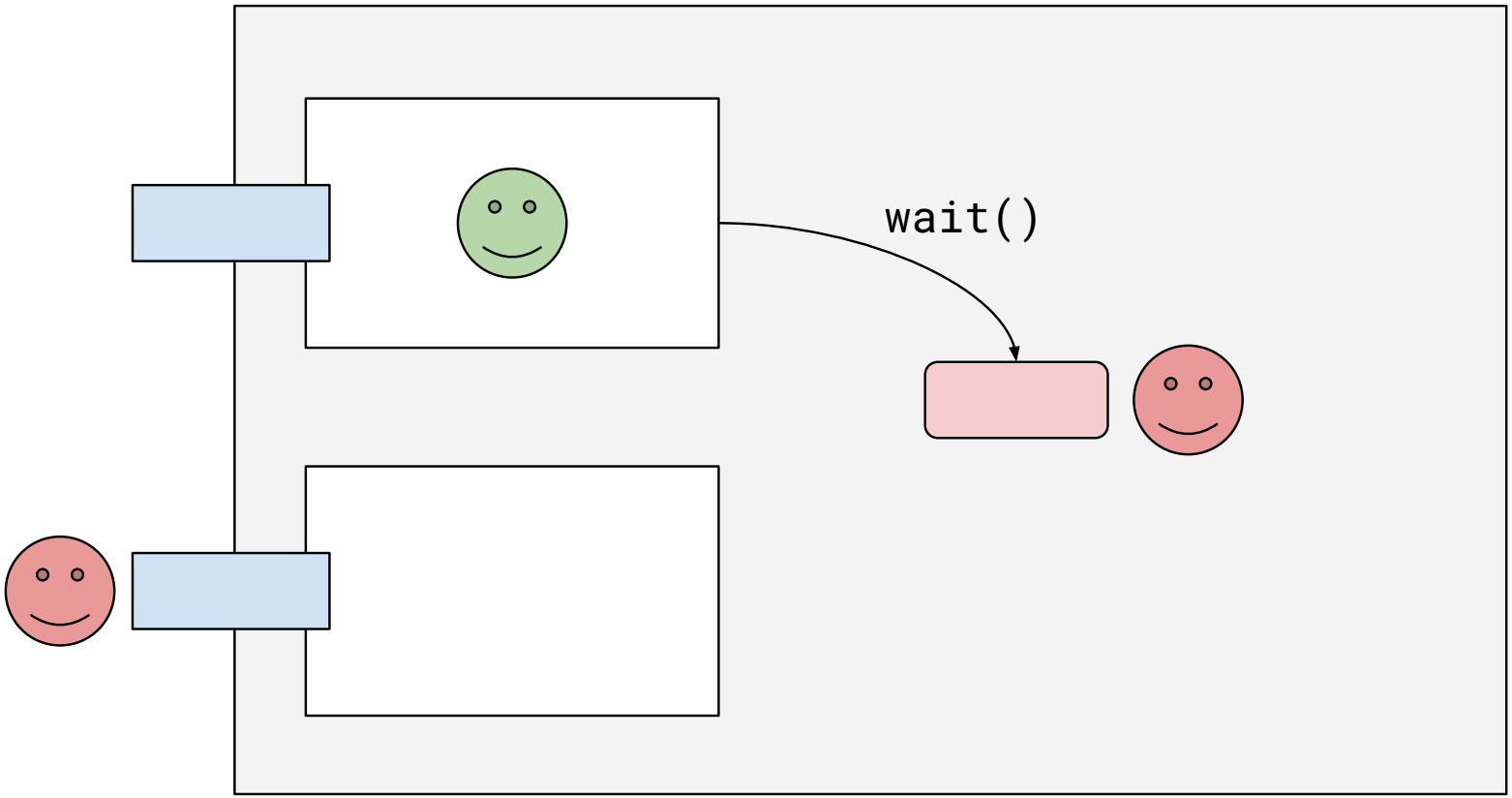
Comportamiento de un monitor (gráficamente)



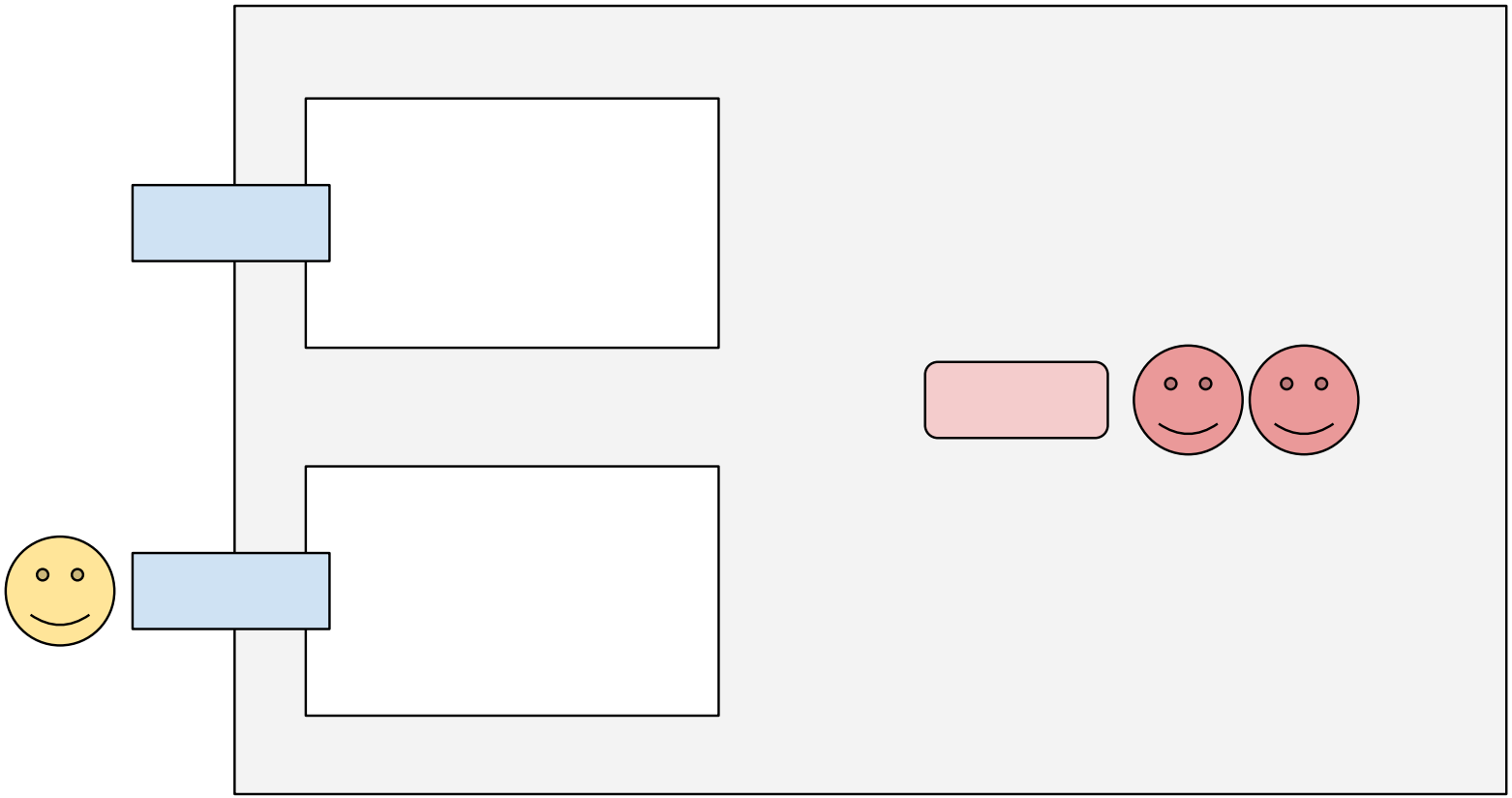
Comportamiento de un monitor (gráficamente)



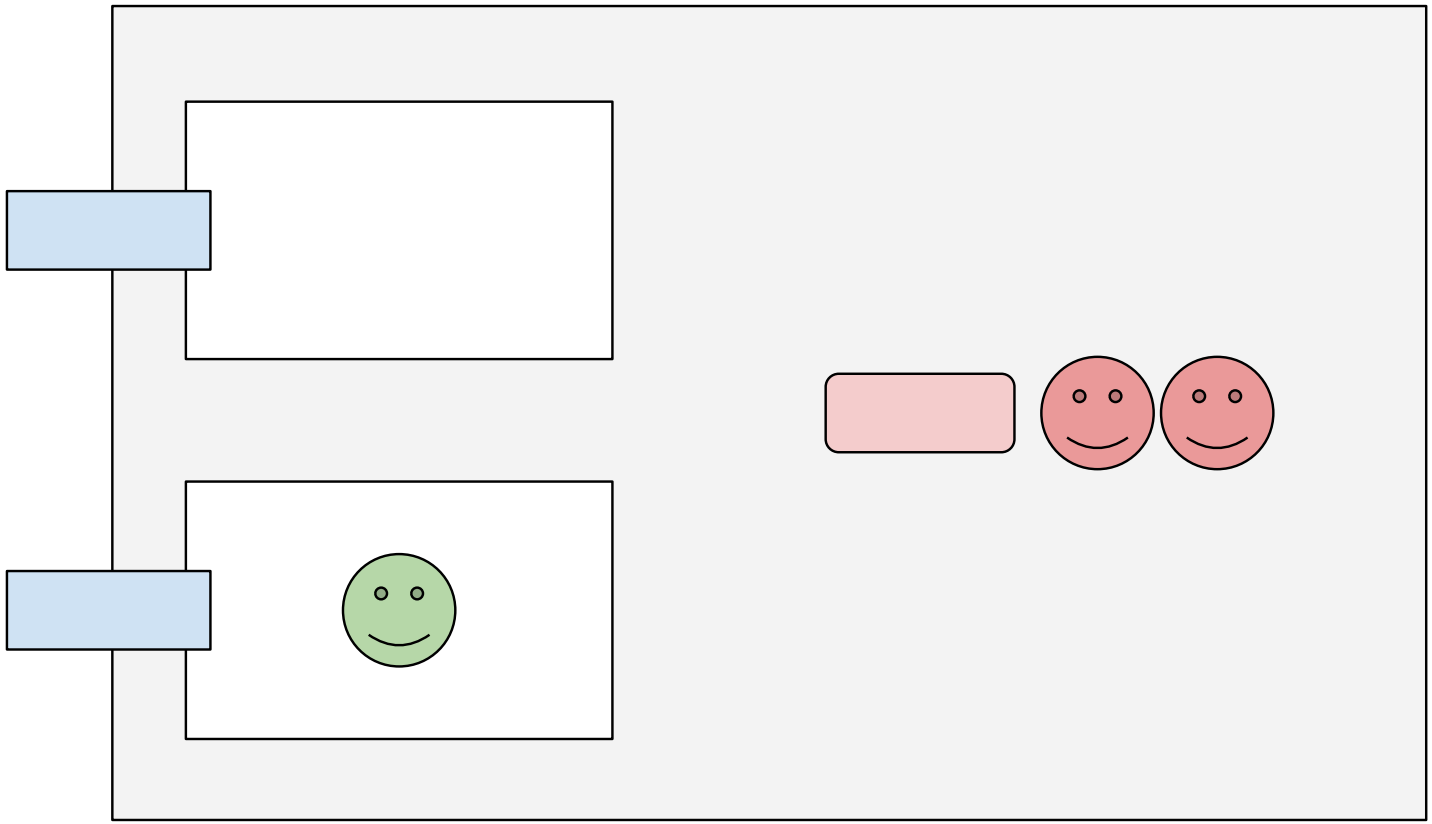
Comportamiento de un monitor (gráficamente)



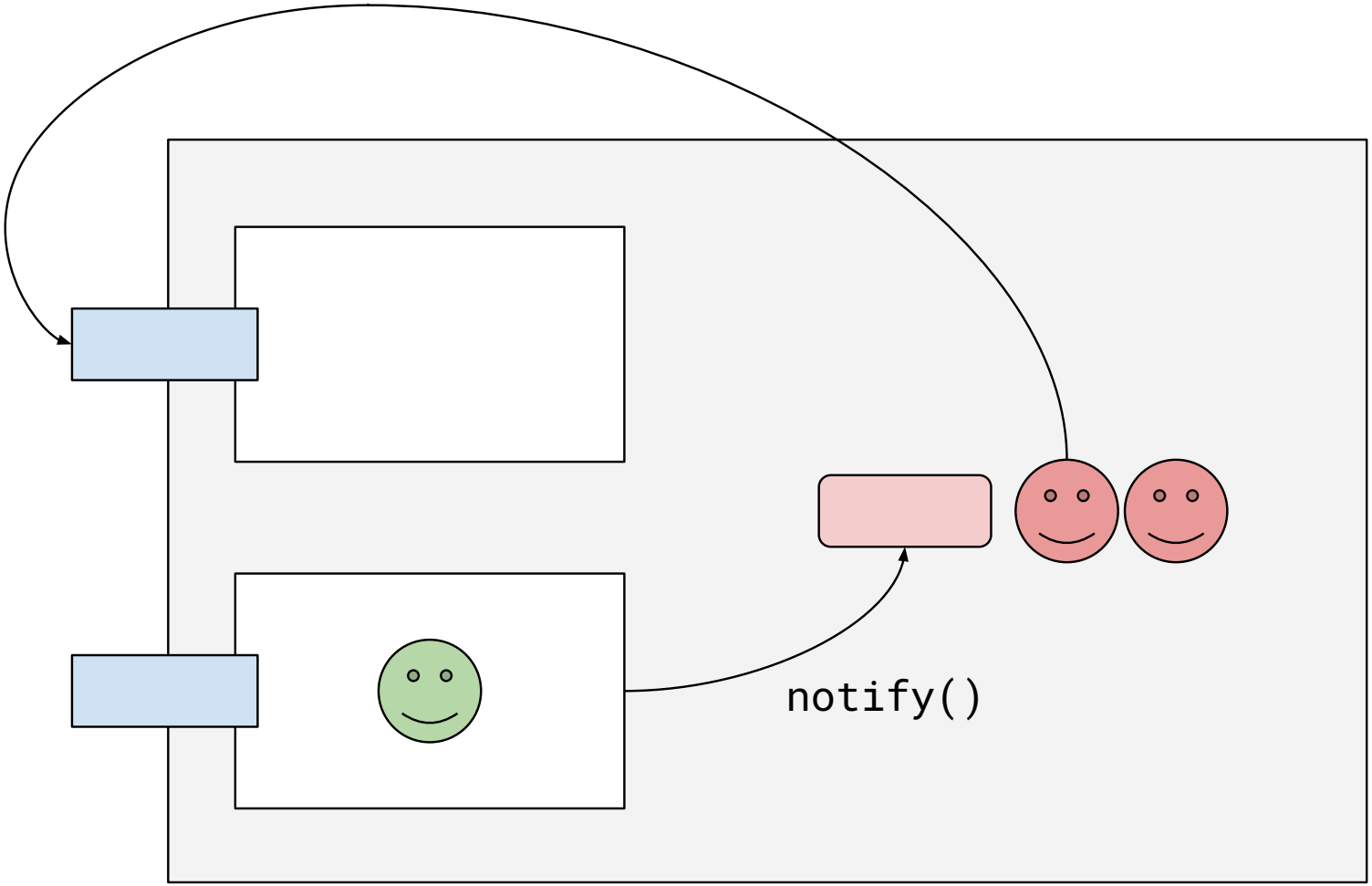
Comportamiento de un monitor (gráficamente)



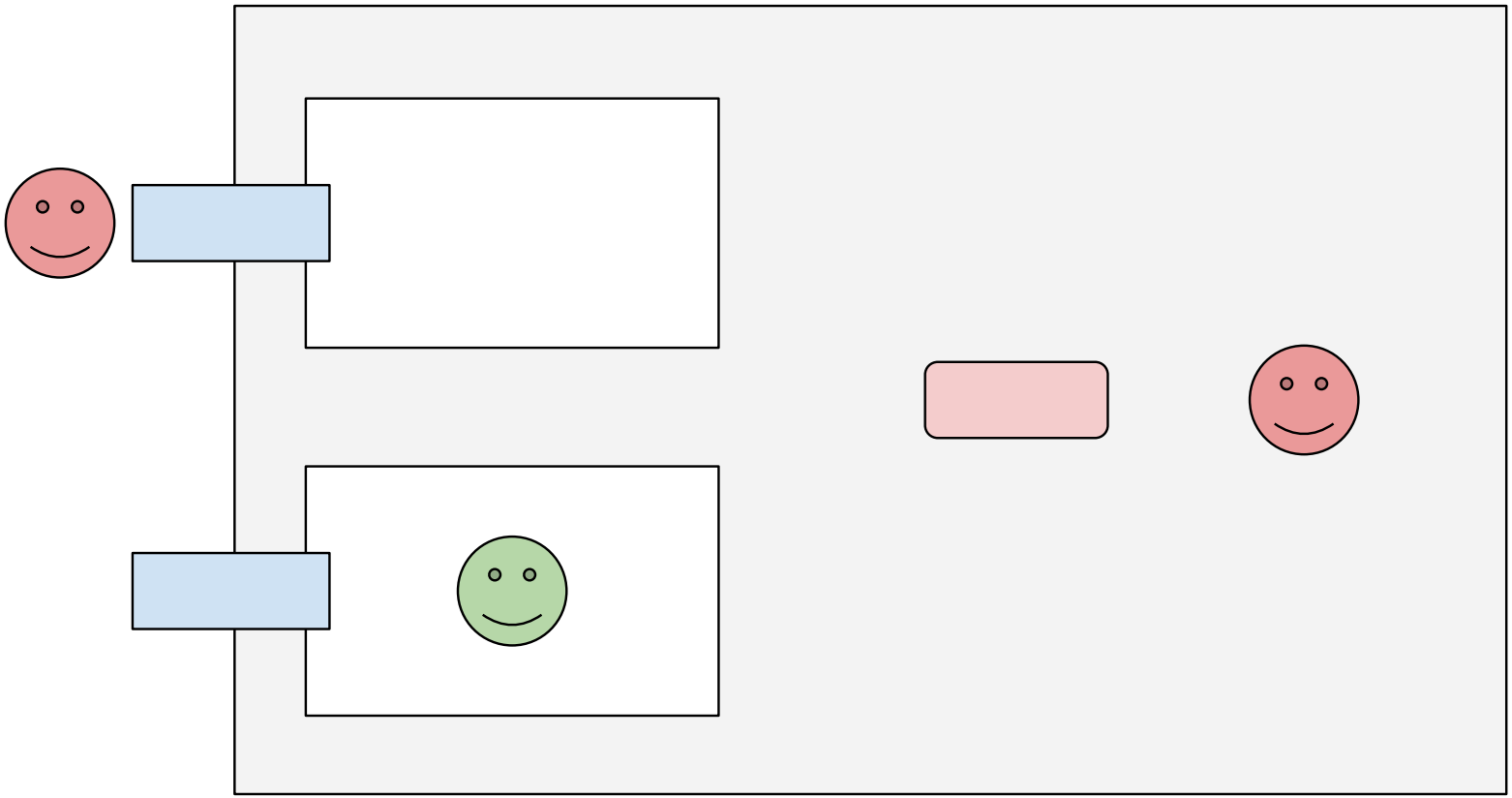
Comportamiento de un monitor (gráficamente)



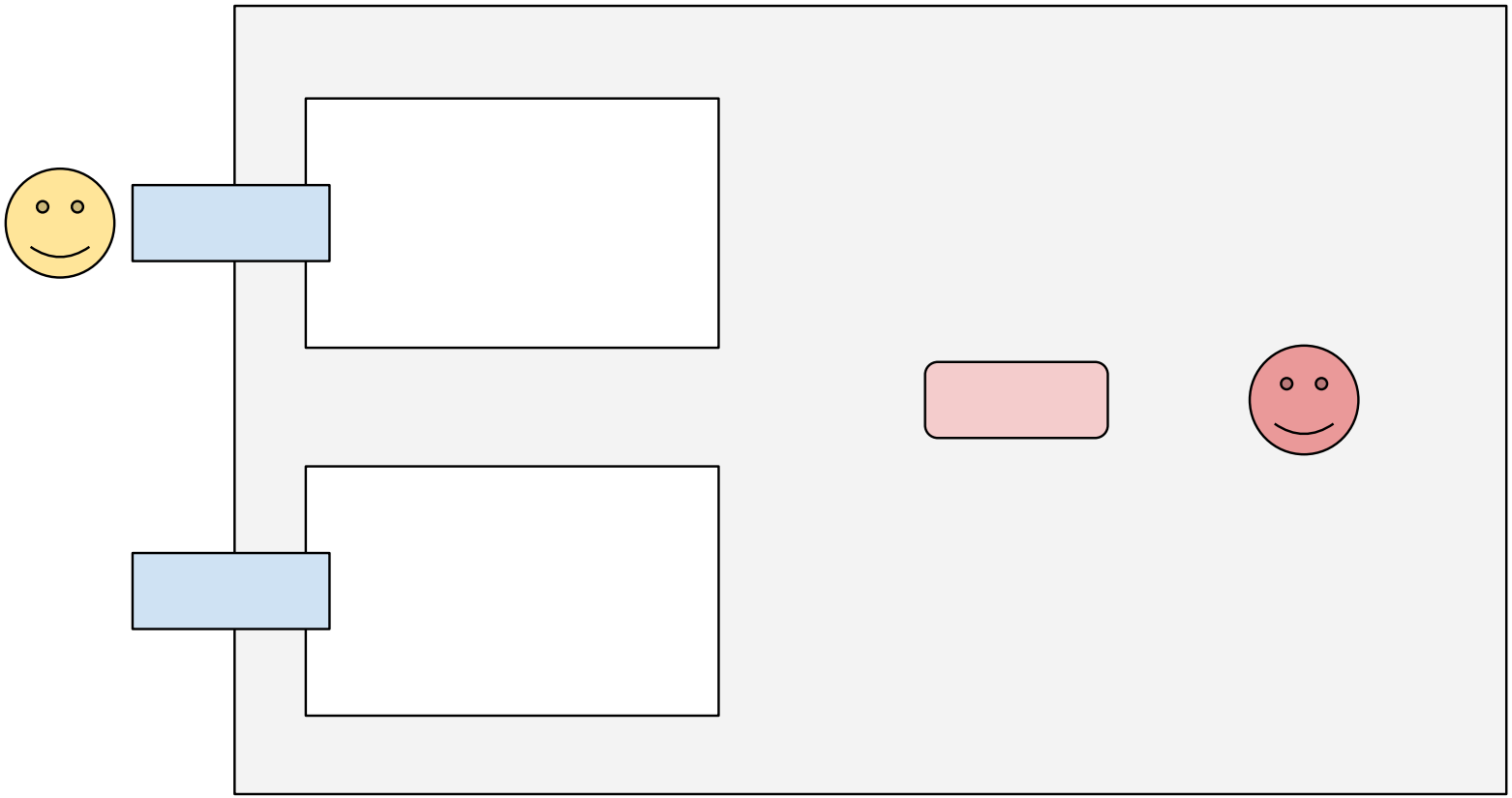
Comportamiento de un monitor (gráficamente)



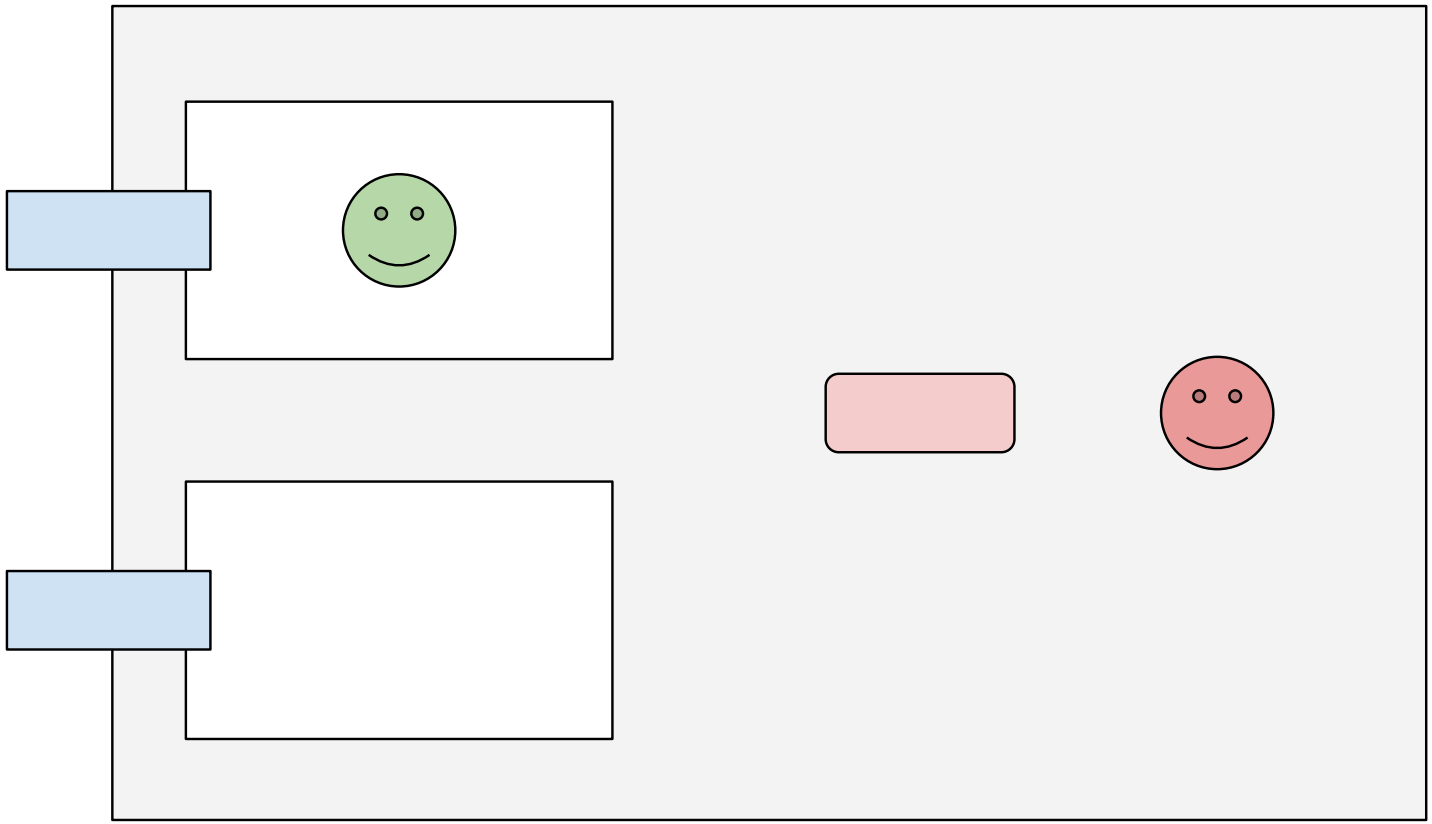
Comportamiento de un monitor (gráficamente)



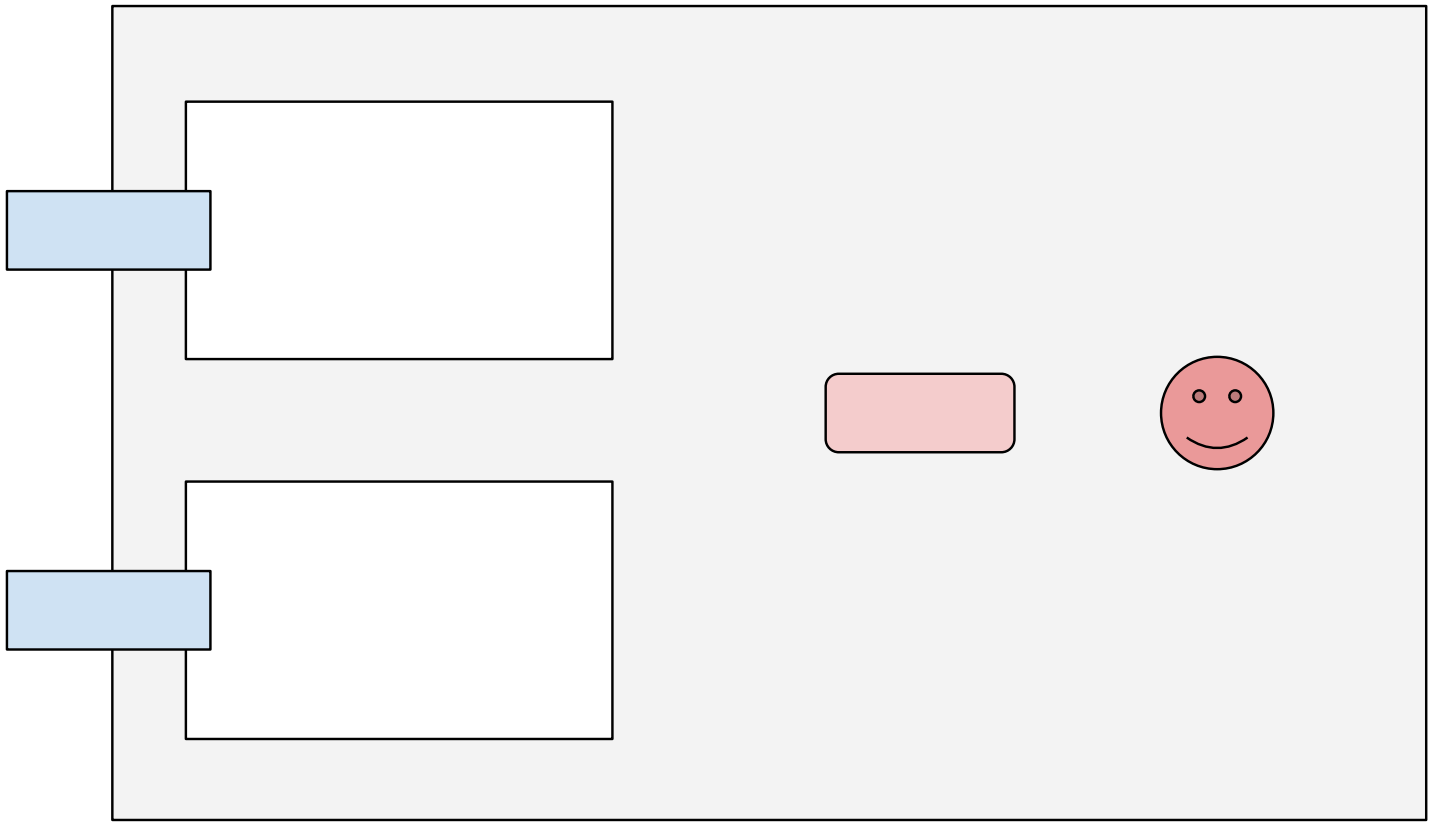
Comportamiento de un monitor (gráficamente)



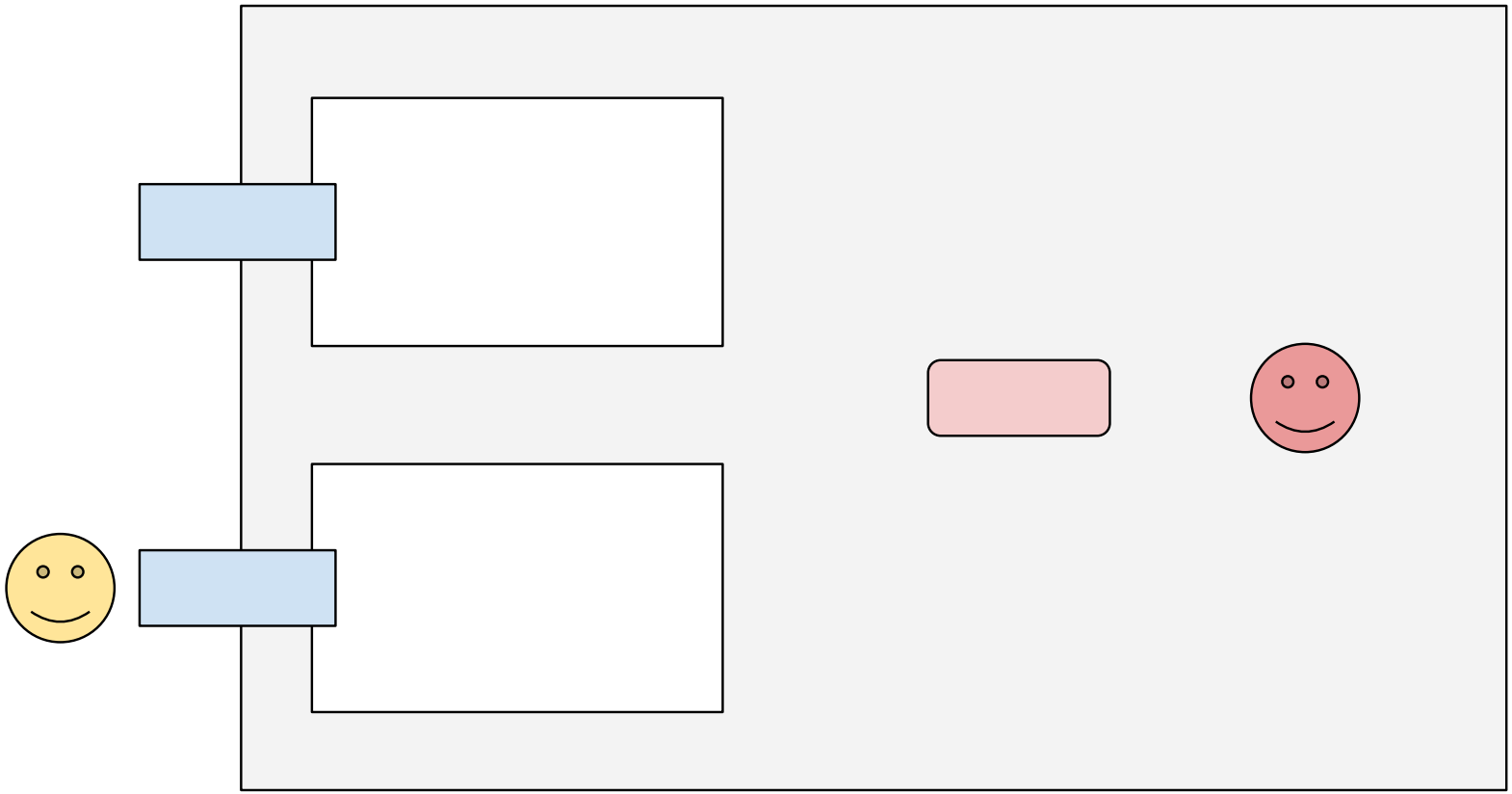
Comportamiento de un monitor (gráficamente)



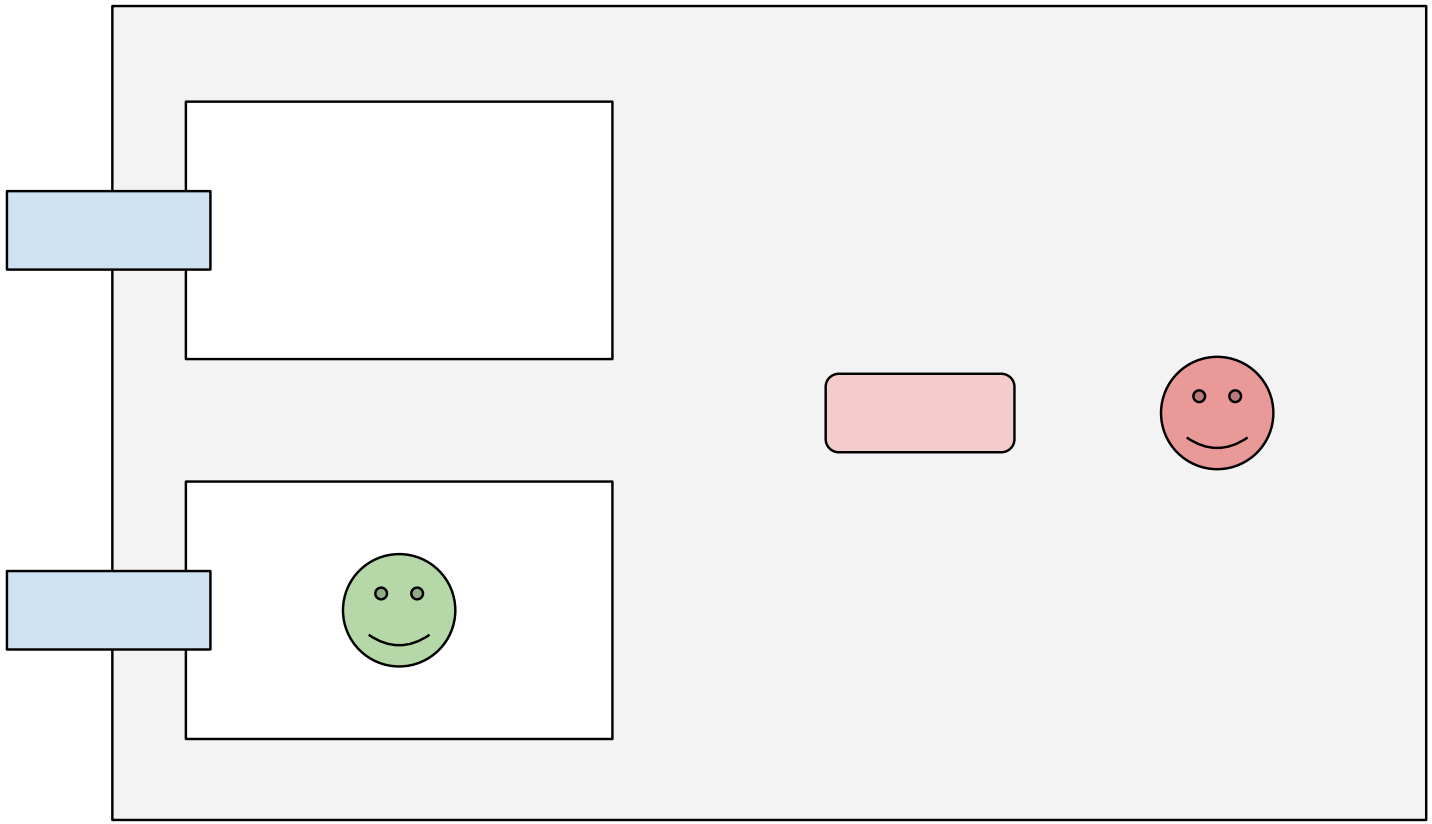
Comportamiento de un monitor (gráficamente)



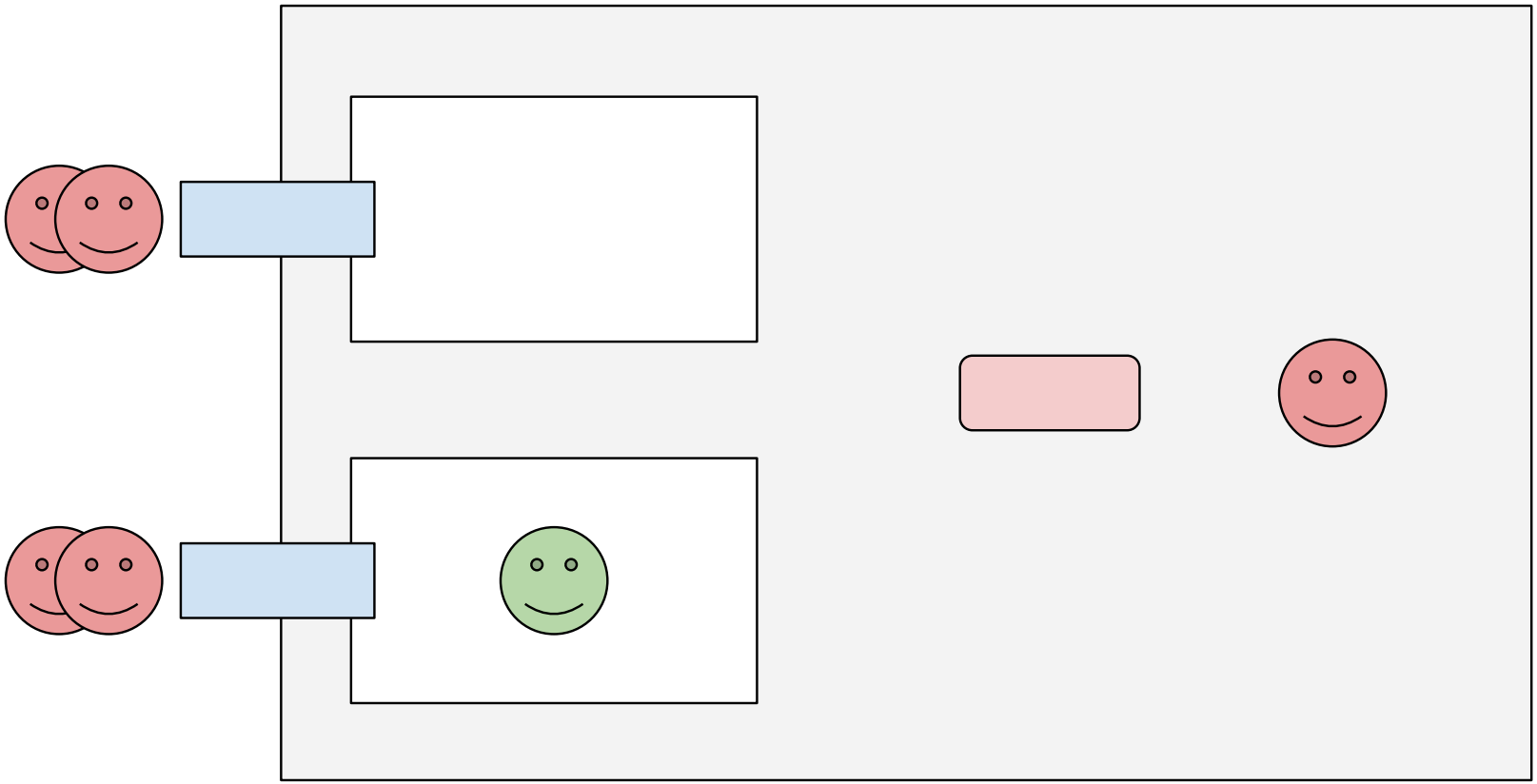
Comportamiento de un monitor (gráficamente)



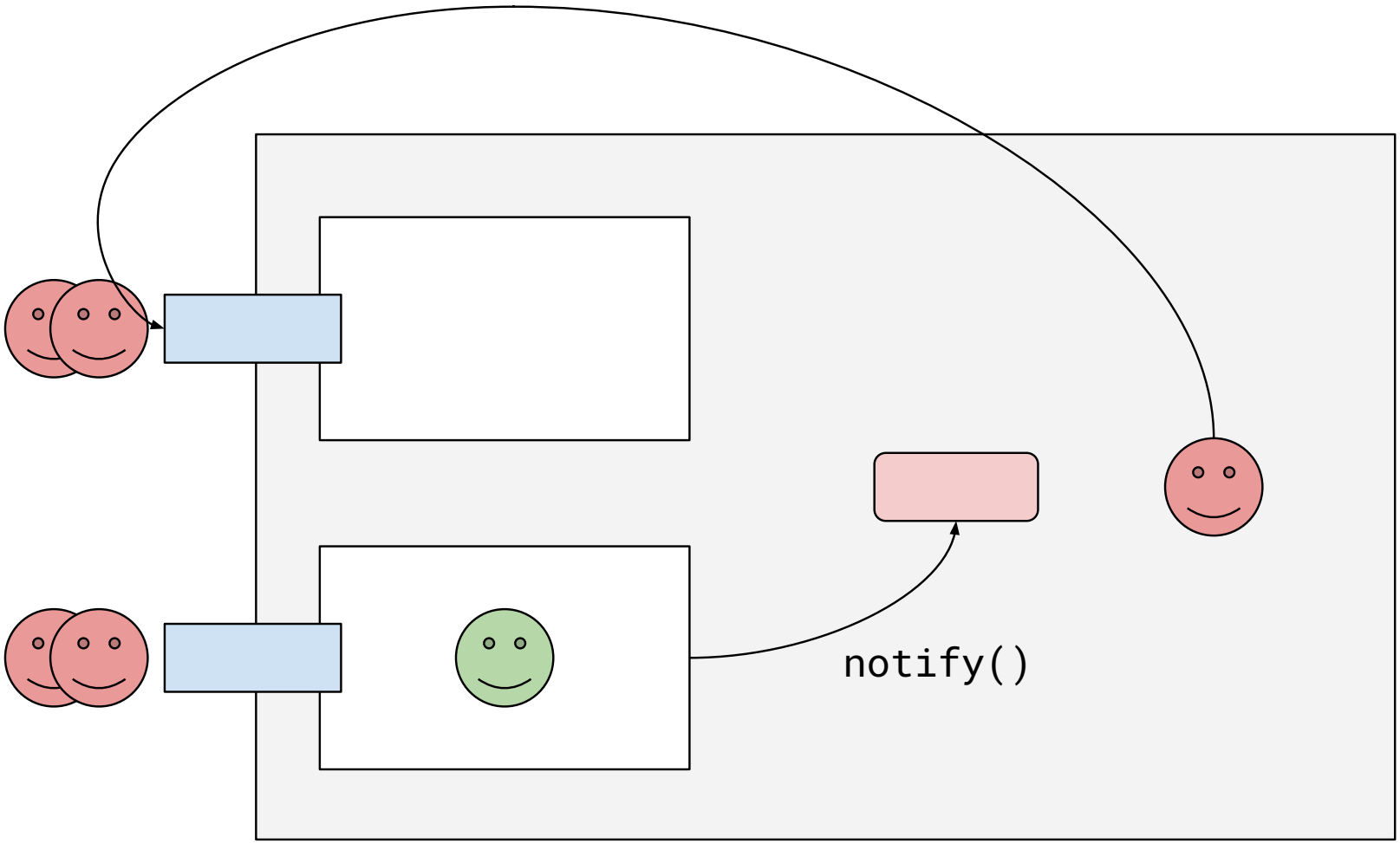
Comportamiento de un monitor (gráficamente)



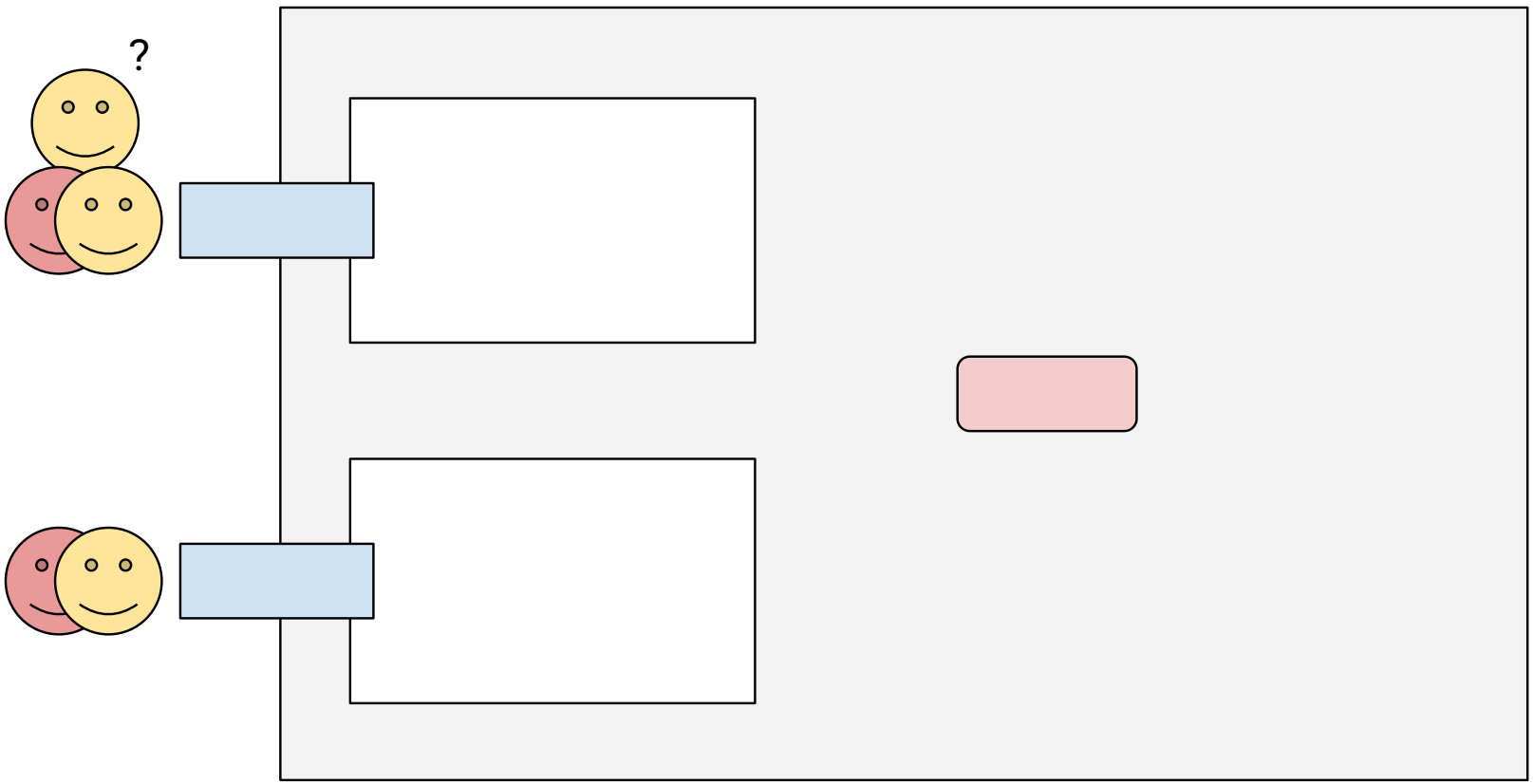
Comportamiento de un monitor (gráficamente)



Comportamiento de un monitor (gráficamente)



Comportamiento de un monitor (gráficamente)



Notify y salida de ejecución

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

Notify y salida de ejecución

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

Notify y salida de ejecución

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

- ¿Qué sucede con el lock?
  - Para continuar la ejecución el proceso desbloqueado debe adquirir el lock.
  - Los threads compiten por el lock, por lo que al adquirirlo tiene que volver checkear la condición por la que se durmió.

Ejemplo: Contador positivo

```
monitor ContadorPositivo {

  private int contador = 0;

  public void incrementar() {
    contador++;
    notify();
  }

  public void decrementar() {
    while (contador == 0) {
      wait();
    }
    contador--;
  }

}
```

Ejemplo: Buffer de dimensión 1

Ejemplo: Buffer de dimensión 1

```
monitor Buffer {

  private Object dato = null;    // el dato compartido

  public Object read() {
    while (dato == null)
      wait();
    aux = dato;
    dato = null;
    notifyAll();
    return aux;
  }

  public void write(Object o) {
    while (dato != null)
      wait();
    dato = o;
    notifyAll();
  }

}
```

Monitores en Java

Monitores en Java

- Toda clase tiene un lock y una única 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).

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 podría mejorar la eficiencia en algunos casos.

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 podría mejorar la eficiencia en algunos casos.

- 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 podría mejorar la eficiencia en algunos casos.
- 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.

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 podría mejorar la eficiencia en algunos casos.
- 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.

IllegalMonitorStateException

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

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

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

InterruptedException

- El método `wait` puede arrojar una excepción.
- Esto ocurre cuando se interrumpe al thread en espera usando el método interrupt.
- Es una buena práctica considerar eventuales interrupciones.

Buffer de dimensión N en Java

Buffer de dimensión N en Java

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

  synchronized void write(Object o) throws InterruptedException {
    while (isFull()) { wait(); }
    data[begin] = o;
    begin = next(begin);
    notifyAll();
  }

  synchronized Object read() throws InterruptedException {
    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); }

}
```

Productor consumidor

Threads en Java (Productor)

```
class Productor extends Thread {

  private Buffer buffer;

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

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

}
```

Threads en Java (Consumidor)

```
class Consumidor implements Runnable {

  private Buffer buffer;

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

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

}
```

Threads en Java (thread principal)

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

  p.start();
  c.start();

}
```

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

Lectores Escritores

Lectores Escritores con Monitores

¿Cuáles deberían ser los métodos para un monitor que representa una base de datos que puede ser leída o escrita?

Lectores Escritores con Monitores

¿Cuáles deberían ser los métodos para un monitor que representa una base de datos que puede ser leída o escrita?

```
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

```
synchronized void beginRead() throws InterruptedException {
  while (!canRead()) {
    wait();
  }
  readers++;
}

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

Lectores Escritores Solución

```
synchronized void beginWrite() throws InterruptedException {
  while (!canWrite()) {
    wait();
  }
  writers = 1;
}

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

Lectores Escritores (Prioridad Escritores)

```
private int waitingWriters = 0;

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

synchronized void beginWrite() throws InterruptedException {
  waitingWriters++;
  while (!canWrite()) {
    wait();
  }
  waitingWriters--;
  writers = 1;
}

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

Resumen

- Con Monitores podemos resolver problemas de sincronización.

Resumen

- Con Monitores podemos resolver problemas de sincronización.
- Tienen un mayor nivel de abstracción que los semáforos.

Resumen

- Con Monitores podemos resolver problemas de sincronización.
- Tienen un mayor nivel de abstracción que los semáforos.
- En Java todo objeto tiene lock y una (única) variable de condición asociada.

Resumen

- Con Monitores podemos resolver problemas de sincronización.
- Tienen un mayor nivel de abstracción que los semáforos.
- En Java todo objeto tiene lock y una (única) variable de condición asociada.
- La "condición" no está en la variable, está en el `while` que envuelve la espera.