Mensajes

Programación concurrente

Hasta ahora vimos:

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

Hasta ahora vimos:

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

¿Cómo lo solucionamos?

¿Cómo lo solucionamos?

Una opción es eliminar la memoria compartida.

¿Cómo lo solucionamos?

Una opción es eliminar la memoria compartida.

En principio esto no sólo elimina el problema, sino que también la posibilidad de comunicar distintos procesos.

Ejemplo

```
process P1:
  // computo costoso
  // en funcion de x
```

```
process P2:
  // computo costoso
  // en funcion de y
```

```
process P3:
  // computo en funcion de los
  // resultados de P1 y P2
```

Definición

Mensajes

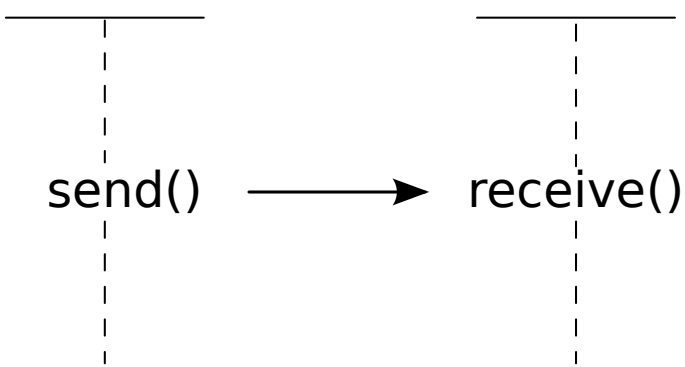
Mecanismo de sincronización que permite enviar y recibir datos a través de un canal de comunicación.

Ejemplo:

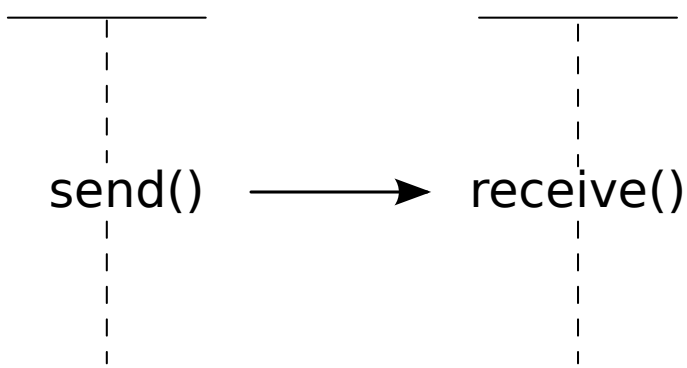
```
canal.send(object);

object = canal.receive();
```

Pensemos en trazas I

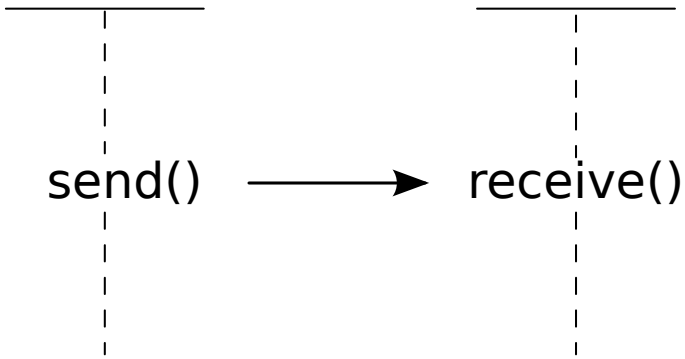


Pensemos en trazas I

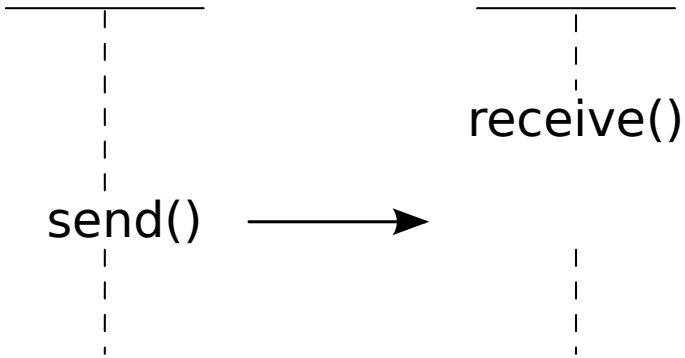


¿Qué sucede si se ejecuta el `receive` antes?

Pensemos en trazas I



¿Qué sucede si se ejecuta el `receive` antes?

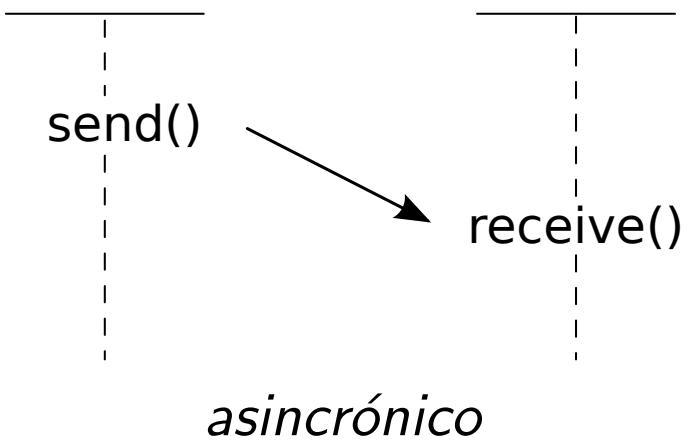


Pensemos en trazas II

¿Qué sucede si se ejecuta el `send` antes?

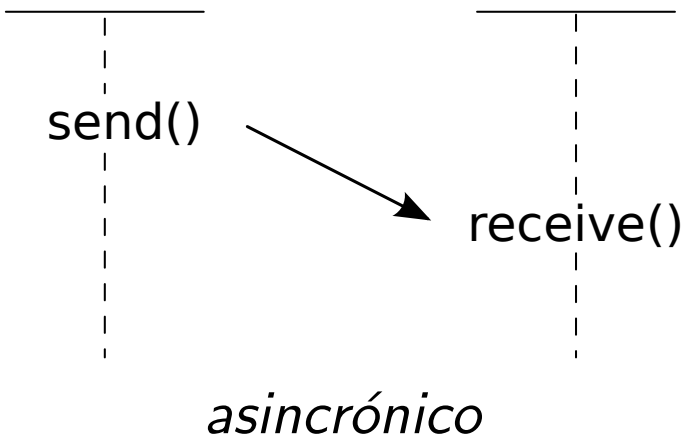
Pensemos en trazas II

¿Qué sucede si se ejecuta el `send` antes?



Pensemos en trazas II

¿Qué sucede si se ejecuta el `send` antes?



Existen variantes sincrónicas (bloqueante) que no estudiaremos.

Definición

Canales

Un canal es un medio de comunicación entre procesos que permite hacer el intercambio de mensajes. Lo veremos como un tipo abstracto de datos ignorando los detalles de su implementación.

Definición

Canales

Un canal es un medio de comunicación entre procesos que permite hacer el intercambio de mensajes. Lo veremos como un tipo abstracto de datos ignorando los detalles de su implementación.

Los podemos ver como buffers

- En teoría, canales asincrónicos pueden verse como buffers de capacidad no acotada.
- En la práctica se implementan con buffers de tamaño acotado.
- Admiten objetos de clases *serializables*.

Identificación de procesos

¿Cómo se identifica el origen y el destino de un mensaje?

Identificación de procesos

¿Cómo se identifica el origen y el destino de un mensaje?

Identificación simétrica directa:

Identificación de procesos

¿Cómo se identifica el origen y el destino de un mensaje?

Identificación simétrica directa:

Identificación asimétrica directa:

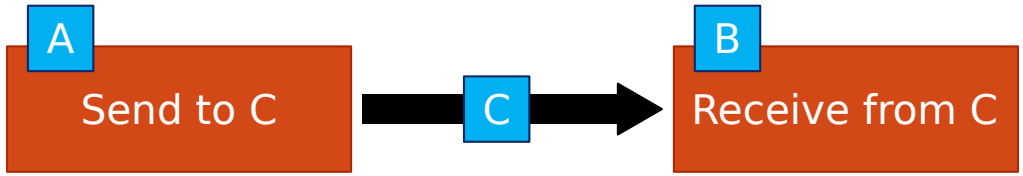
Identificación de procesos

¿Cómo se identifica el origen y el destino de un mensaje?

Identificación simétrica directa:

Identificación asimétrica directa:

Identificación indirecta:



Retomando el ejemplo I

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();

process P1: {
  // computo costoso
  // en funcion de un X
  c1.send(r1);
}
```

```
process P2: {
  // computo costoso
  // en funcion de un Y
  c2.send(r2);
}
```

```
process P3: {
  r1 = c1.receive();
  r2 = c2.receive();
  print(r1 + r2);
}
```

Retomando el ejemplo II

¿Cómo hacemos si X e Y están inicialmente in P3?

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();

process P1: {
  x = c1.receive();
  // computo costoso
  // en funcion de x
  c1.send(r1);
}
```

```
process P2: {
  y = c2.receive();
  // computo costoso
  // en funcion de y
  c2.send(r2);
}
```

```
process P3: {
  c1.send(X);
  c2.send(Y);
  r1 = c1.receive();
  r2 = c2.receive();
  print(r1 + r2);
}
```

¿Qué pasa con canales **asincrónicos**?

```
process P3: {
  c1.send(X);
  c2.send(Y);
  r1 = c1.receive();
  r2 = c2.receive();
  print(r1 + r2);
}
```

¿Qué pasa con canales **asincrónicos**?

```
process P3: {
  c1.send(X);
  c2.send(Y);
  r1 = c1.receive();
  r2 = c2.receive();
  print(r1 + r2);
}
```

Puede darse la siguiente secuencia de eventos

1. P3 envía X por c1
2. P3 envía Y por c2
3. P3 lee X de c1 (P1 todavía no lo leyó)
4. P3 realiza el computo del resultado usando X en lugar de r1
5. Error!

Retomando el ejemplo II

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();
global Channel c3 = new Channel();
global Channel c4 = new Channel();

process P1: {
  x = c1.receive();
  // computo costoso
  // en funcion de x
  c3.send(r1);
}
```

```
process P2: {
  y = c2.receive();
  // computo costoso
  // en funcion de y
  c4.send(r2);
}
```

```
process P3: {
  c1.send(X);
  c2.send(Y);
  r1 = c3.receive();
  r2 = c4.receive();
  print(r1 + r2);
}
```

Pregunta

¿Qué diferencia hay entre estas dos formas de escribir P3?

```
process P3: {
  c1.send(X);
  c2.send(Y);
  r1 = c3.receive();
  r2 = c4.receive();
  print(r1 + r2);
}
```

```
process P3: {
  c1.send(X);
  r1 = c3.receive();
  c2.send(Y);
  r2 = c4.receive();
  print(r1 + r2);
}
```

Pregunta

¿Qué diferencia hay entre estas dos formas de escribir P3?

```
process P3: {
  c1.send(X);
  c2.send(Y);
  r1 = c3.receive();
  r2 = c4.receive();
  print(r1 + r2);
}
```

```
process P3: {
  c1.send(X);
  r1 = c3.receive();
  c2.send(Y);
  r2 = c4.receive();
  print(r1 + r2);
}
```

La segunda no aprovecha la concurrencia dado que espera a recibir el resultado del proceso P1 antes de iniciar el cómputo del proceso P2.

Ejercicios

- Escribir dos procesos A y B, el primero genera un número aleatorio y se lo envía al segundo que al recibirlo lo muestra por pantalla.

Ejercicios

- Escribir dos procesos A y B, el primero genera un número aleatorio y se lo envía al segundo que al recibirlo lo muestra por pantalla.

```
global Channel c1 = new Channel();

process A: {
  Integer r =
    new Integer(randint());
  c1.send(r);
}
```

```
process B: {
  Integer r = c1.receive();
  print(r);
}
```

Ejercicios

- Extender el ejercicio anterior para que el segundo proceso lea un número introducido por el usuario que indique la cantidad de números aleatorios a generar. Ese valor debe ser enviado al primer proceso quien debe generar la cantidad de números pedida. El segundo proceso debe mostrar todos los números por pantalla.

Ejercicios

- Extender el ejercicio anterior para que el segundo proceso lea un número introducido por el usuario que indique la cantidad de números aleatorios a generar. Ese valor debe ser enviado al primer proceso quien debe generar la cantidad de números pedida. El segundo proceso debe mostrar todos los números por pantalla.

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();

process A: {
  int n = c2.receive();
  repeat (n) {
    Integer r =
      new Integer(randint());
    c1.send(r);
  }
}
```

```
process B: {
  int n = parseInt(read());
  c2.send(n);
  repeat (n) {
    Integer r = c1.receive();
    print(r);
  }
}
```

Pregunta

¿Cómo podemos extender el ejercicio anterior para admitir pedidos concurrentes (desde múltiples instancias de B)?

Pregunta

¿Cómo podemos extender el ejercicio anterior para admitir pedidos concurrentes (desde múltiples instancias de B)?

- Necesitamos un proceso que reciba el pedido y delegue en otro la generación de los números.

Pregunta

¿Cómo podemos extender el ejercicio anterior para admitir pedidos concurrentes (desde múltiples instancias de B)?

- Necesitamos un proceso que reciba el pedido y delegue en otro la generación de los números.

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();

process A: {
  while (true) {
    int n = c2.receive();
    thread (n)
      repeat (n) {
        Integer r = new Integer(
          randint());
        c1.send(r);
      }
  }
}
```

```
process B: {
  int n = parseInt(read());
  c2.send(n);
  repeat (n) {
    Integer r = c1.receive();
    print(r);
  }
}
```

Inicialización de Threads

- Tenemos la posibilidad de iniciar un thread nuevo en nuestro proceso si lo necesitamos

Inicialización de Threads

- Tenemos la posibilidad de iniciar un thread nuevo en nuestro proceso si lo necesitamos
- Si lo hacemos en Java o Hydra, esto nos dará acceso a variables compartidas

Inicialización de Threads

- Tenemos la posibilidad de iniciar un thread nuevo en nuestro proceso si lo necesitamos
- Si lo hacemos en Java o Hydra, esto nos dará acceso a variables compartidas
- En esta parte de la materia buscamos no utilizar la memoria compartida para comunicar threads

Inicialización de Threads

- Tenemos la posibilidad de iniciar un thread nuevo en nuestro proceso si lo necesitamos
- Si lo hacemos en Java o Hydra, esto nos dará acceso a variables compartidas
- En esta parte de la materia buscamos no utilizar la memoria compartida para comunicar threads
- Por ello, vamos a usar la notación de poner un "parámetro" a los threads indicando variables que se **copian** en la creación de un thread nuevo

Volviendo al Problema

¿Cómo extendemos el ejercicio para enviar números aleatorios entre 0 y un número provisto por el cliente?

Volviendo al Problema

¿Cómo extendemos el ejercicio para enviar números aleatorios entre 0 y un número provisto por el cliente?
Contaremos con una clase `Request` de campos variables

Volviendo al Problema

¿Cómo extendemos el ejercicio para enviar números aleatorios entre 0 y un número provisto por el cliente?
Contaremos con una clase `Request` de campos variables

```
  global Channel c1 = new Channel();
  global Channel c2 = new Channel();

process A: {
  while (true) {
    req = c1.receive();
    thread (req)
      repeat (req.n) {
        Integer r = new Integer(
          random() * req.m));
        c2.send(r);
      }
  }
}
```

```
process B: {
  req = new Request();
  req.n = parseInt(read());
  req.m = parseInt(read());
  c1.send(req);
  repeat (req.n)
    print(c2.receive());
}
```

Volviendo al Problema

¿Cómo extendemos el ejercicio para enviar números aleatorios entre 0 y un número provisto por el cliente?
Contaremos con una clase `Request` de campos variables

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();

process A: {
  while (true) {
    req = c1.receive();
    thread (req)
      repeat (req.n) {
        Integer r = new Integer(
          random() * req.m));
        c2.send(r);
      }
  }
}
```

```
process B: {
  req = new Request();
  req.n = parseInt(read());
  req.m = parseInt(read());
  c1.send(req);
  repeat (req.n)
    print(c2.receive());
}
```

¿Está bien esta solución?

Volviendo al Problema

¿Cómo extendemos el ejercicio para enviar números aleatorios entre 0 y un número provisto por el cliente?
Contaremos con una clase `Request` de campos variables

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();

process A: {
  while (true) {
    req = c1.receive();
    thread (req)
      repeat (req.n) {
        Integer r = new Integer(
          random() * req.m));
        c2.send(r);
      }
  }
}
```

```
process B: {
  req = new Request();
  req.n = parseInt(read());
  req.m = parseInt(read());
  c1.send(req);
  repeat (req.n)
    print(c2.receive());
}
```

¿Está bien esta solución? No, algún B puede recibir números fuera de su rango.

Solución

¿Cómo lo solucionamos?

Solución

¿Cómo lo solucionamos?

- Necesitamos un canal de respuesta para cada cliente.

Solución

¿Cómo lo solucionamos?

- Necesitamos un canal de respuesta para cada cliente.
- Dependiendo del protocolo el canal puede proponerlo el cliente o elegirlo el servidor.

Solución

¿Cómo lo solucionamos?

- Necesitamos un canal de respuesta para cada cliente.
- Dependiendo del protocolo el canal puede proponerlo el cliente o elegirlo el servidor.

```
global Channel c1 = new Channel();

process A: {
  while (true) {
    req = c1.receive();
    thread (req)
      repeat (req.n) {
        Integer r = new Integer(
          random() * req.m);
        req.c.send(r);
      }
  }
}
```

```
process B: {
  req = new Request();
  req.n = parseInt(read());
  req.m = parseInt(read());
  req.c = new Channel();
  c1.send(req);
  repeat (req.n)
    print(req.c.receive());
}
```

Resumen

Resumen

1. Los canales permiten resolver los problemas de concurrencia.

Resumen

1. Los canales permiten resolver los problemas de concurrencia.

2. Trabajan en un esquema sin memoria compartida.

Resumen

1. Los canales permiten resolver los problemas de concurrencia.
2. Trabajan en un esquema sin memoria compartida.
3. La topología puede variar en tiempo de ejecución.

Actor Model

Propuesta de Carl Hewit en 1973 (y refinado por muchos otros).

Actor

Un Actor es una entidad computacional que, en respuesta a un mensaje que recibe, puede:

- enviar mensajes a otros actores que conoce;
- crear nuevos actores; y
- cambiar su comportamiento ante la recepción del próximo mensaje.

Cada actor tiene un canal asociado que lo identifica y por donde recibe mensajes. Por lo que sólo se puede comunicar con otros Actores que conoce o crea. Un Actor puede conocer a otro si recibe su canal como parte de un mensaje.