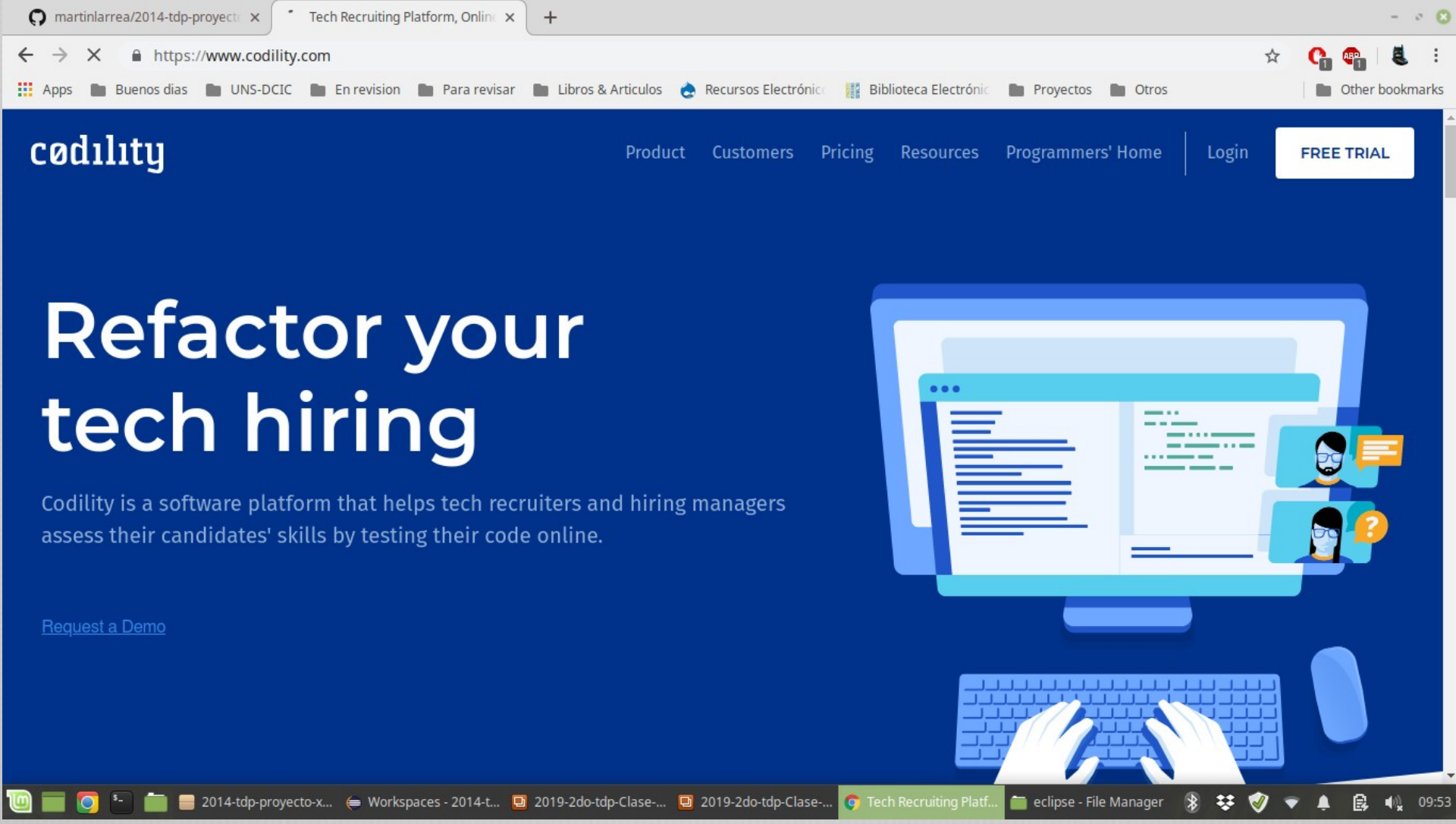
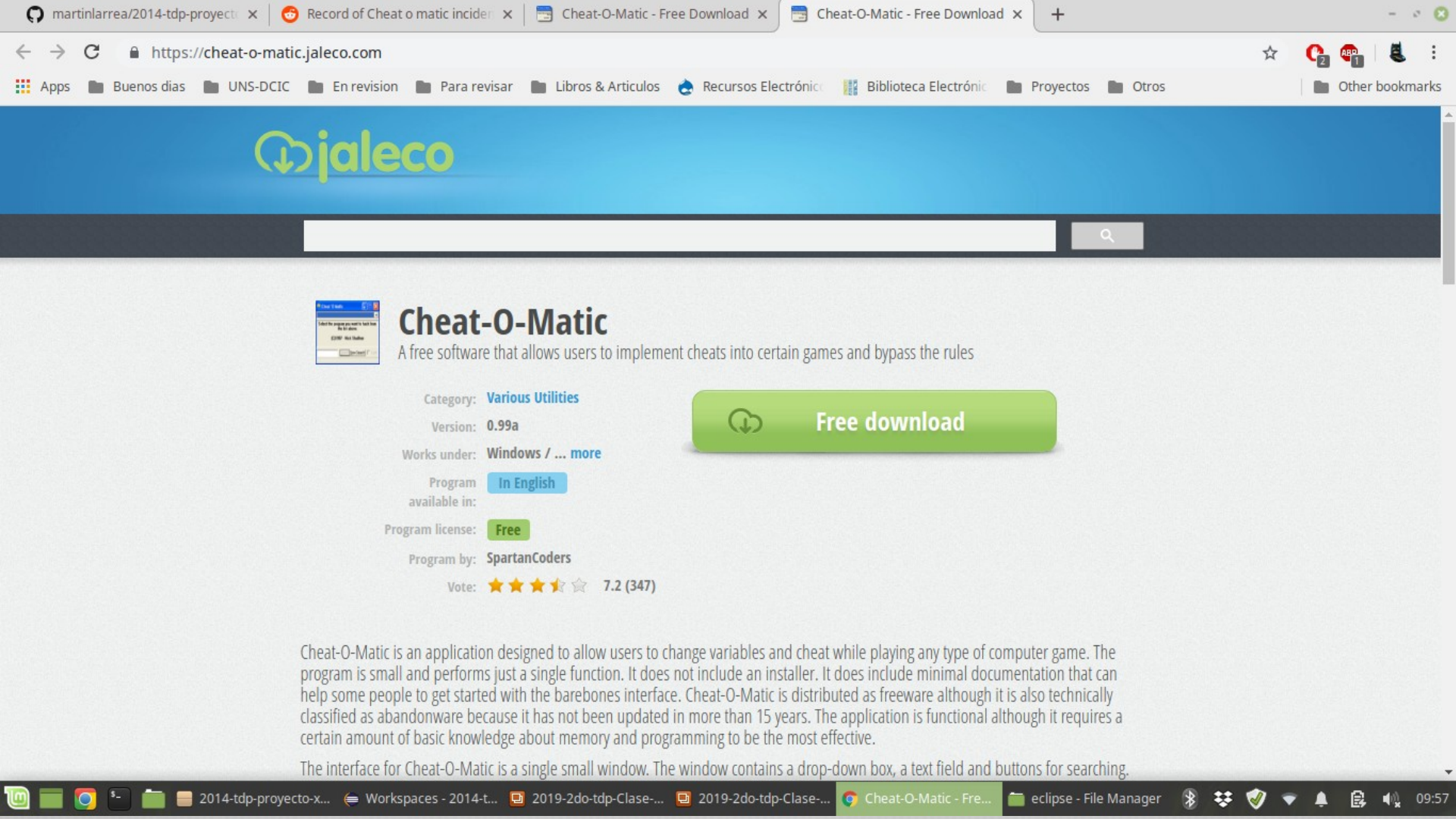
Tecnología de Programación

Martín L. Larrea

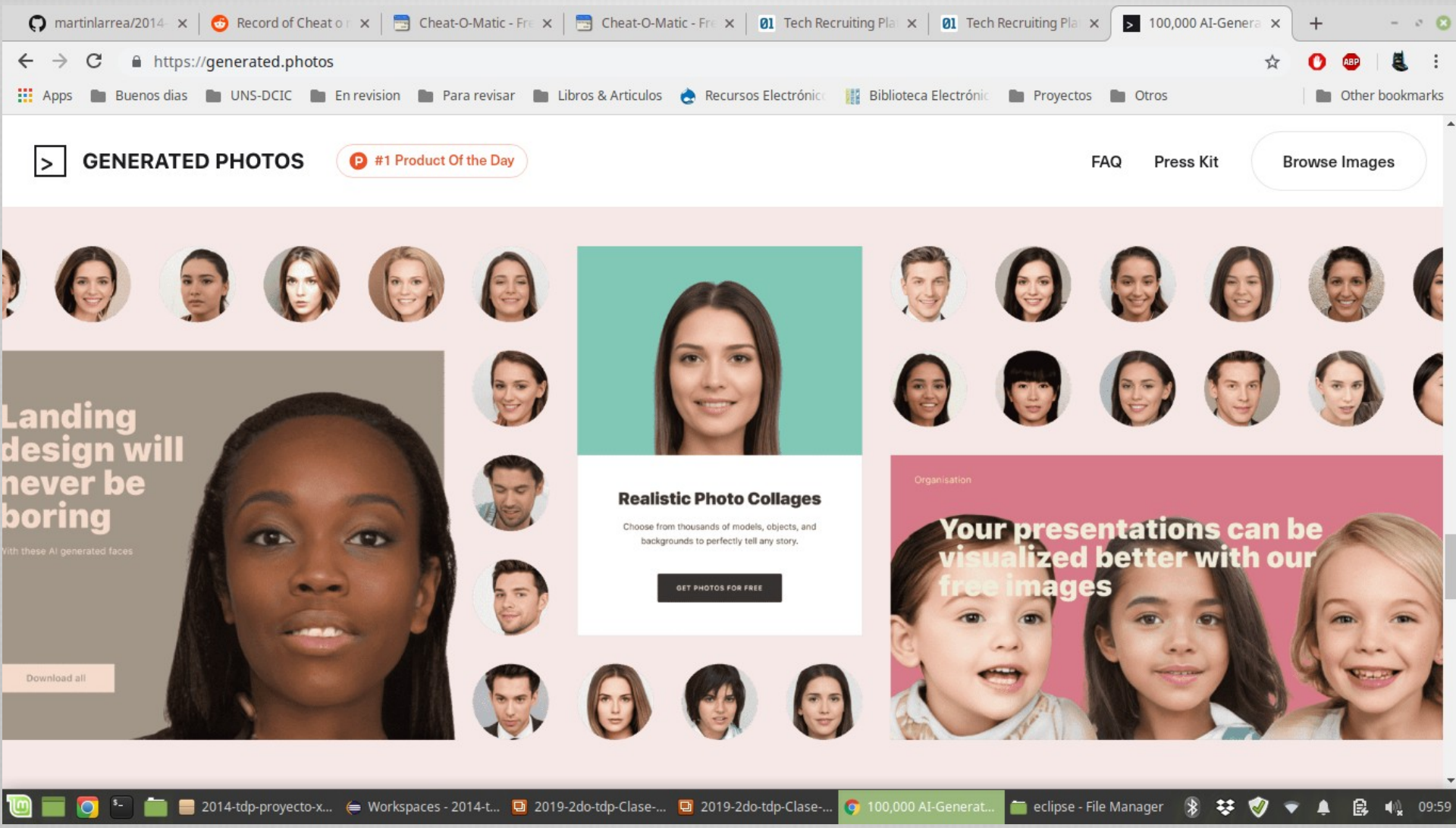
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur



www.codility.com



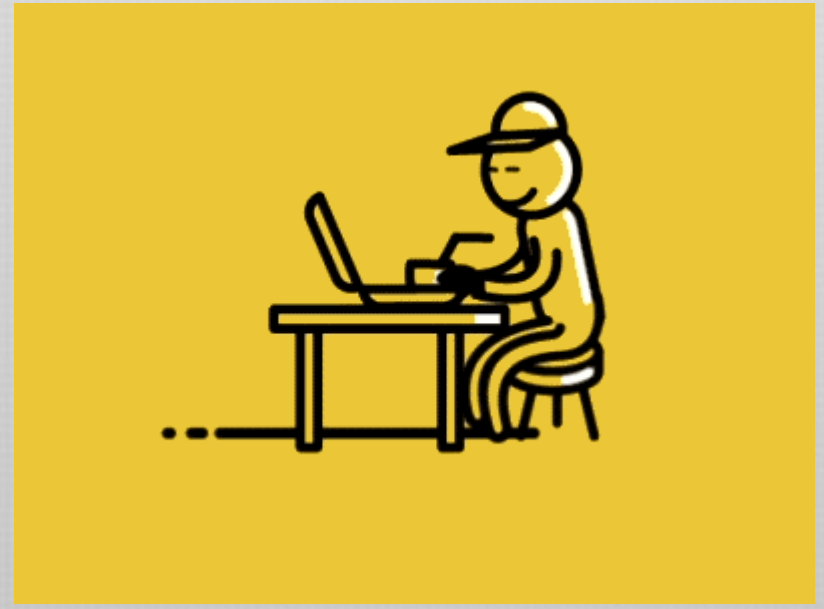
Cheat-O-Matic



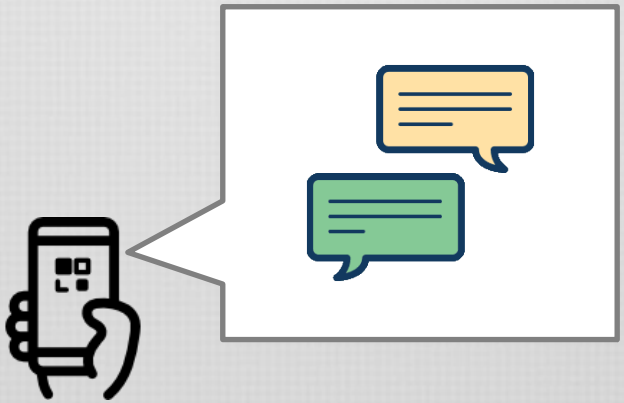
https://generated.photos/

Introducción

Recordemos que entre los objetivos del curso
está perfeccionarnos en
aspectos técnicos de programación,
no solo de diseño



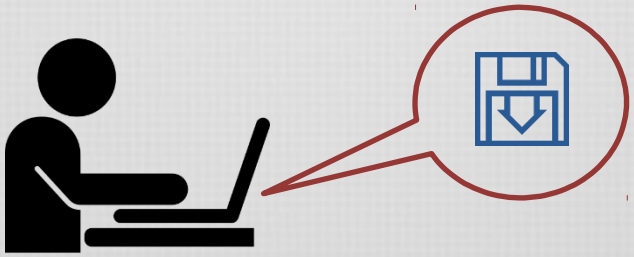
Introducción



Simultáneamente

...se permite escribir texto
... se verifican nuevos mensajes
... se descargan imágenes y videos

Introducción



Espontáneamente
(no necesariamente en respuesta a algún evento)

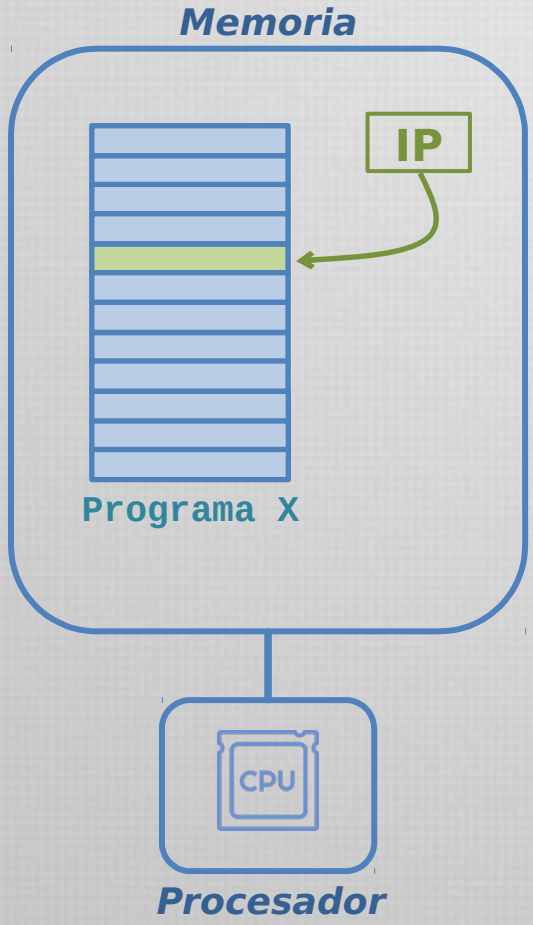
... se guarda el documento
... se hace un backup de los datos

Introducción

Continuamente o periódicamente

... se actualizan posiciones de los jugadores
... se acepta el input del usuario
... se mueve la cámara
... se muestran notificaciones del juego
... se activan animaciones

Programas secuenciales



Un solo programa en memoria.

Un indicador (*instruction pointer*) marca la próxima sentencia a ejecutar.

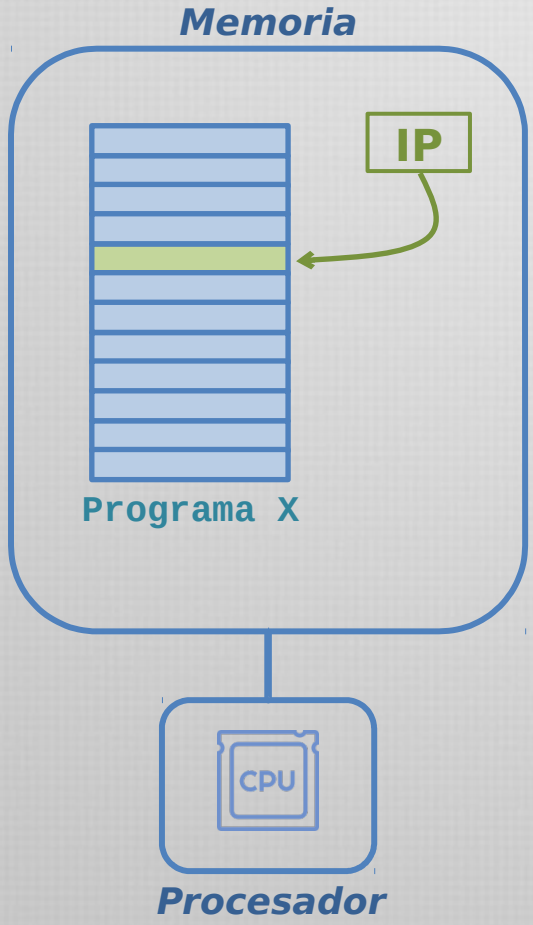
El *instruction pointer* “avanza” secuencialmente o de acuerdo a instrucciones específicas.

Un programa secuencial tiene entonces
un sólo hilo de control (*thread*)

Su ejecución se denomina proceso.

En general,
un proceso es la ejecución particular de un programa particular.

Concurrencia



Sin embargo...

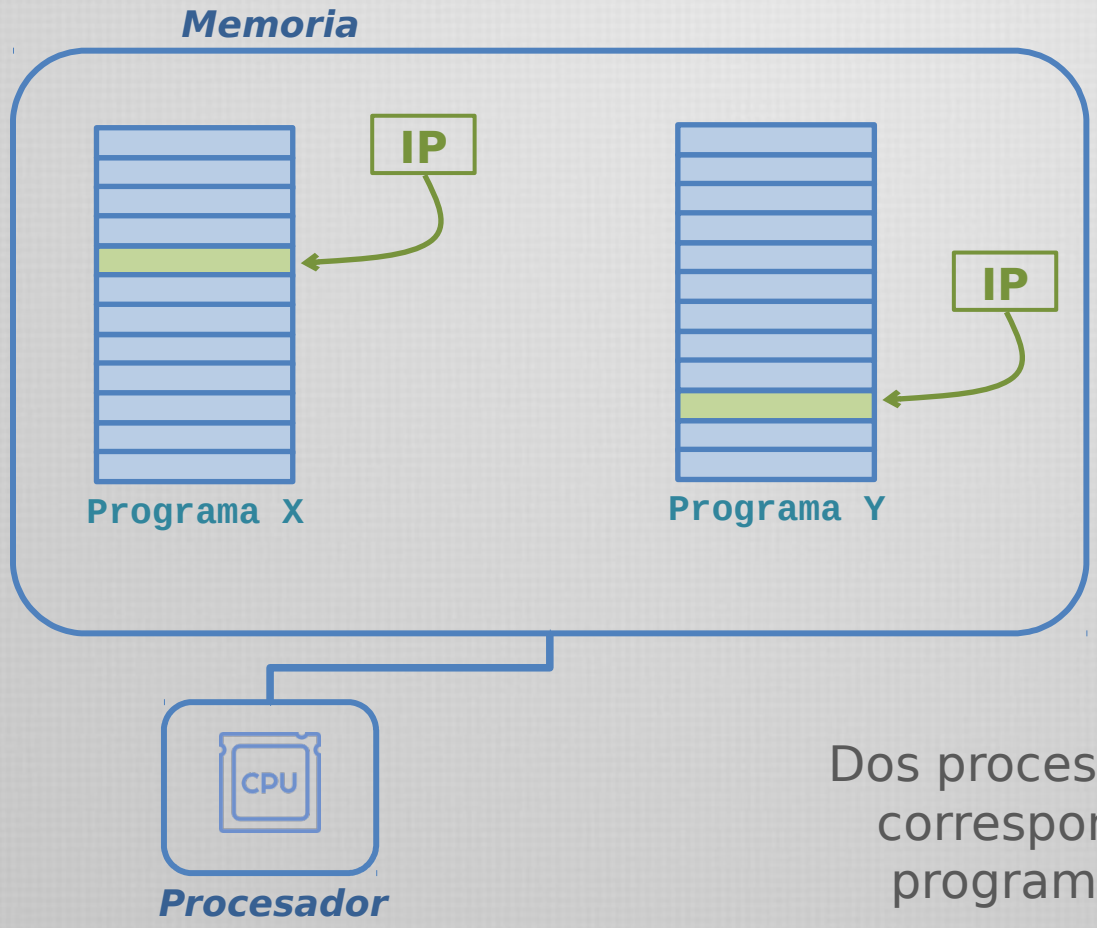
Algunos programas deben “esperar” que ocurran eventos (como entrada y salida de datos). Mientras tanto, no hacen nada!

A veces es mas fácil escribir pequeños programas que realicen tareas simples en forma coordinada.

Es menos “realista”: en el mundo realizamos
más de una tarea a la vez...

Los sistemas operativos han evolucionado para permitir ejecutar más de un programa a la vez, administrando más de un proceso.

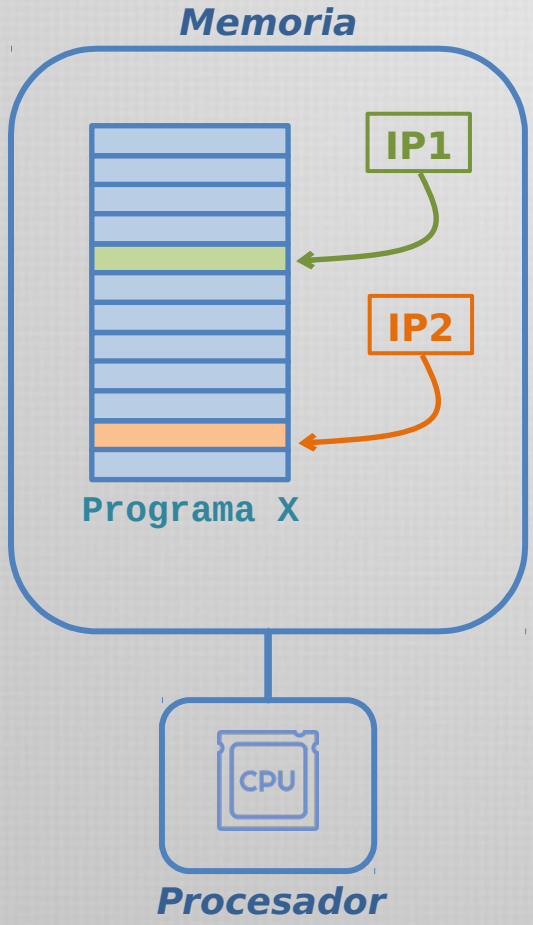
Procesos concurrentes



Dos procesos en ejecución, correspondientes a dos programas diferentes.

Existe un *instruction pointer para cada uno de ellos.*

Procesos concurrentes



Dos procesos en ejecución, correspondientes a un mismo programa.

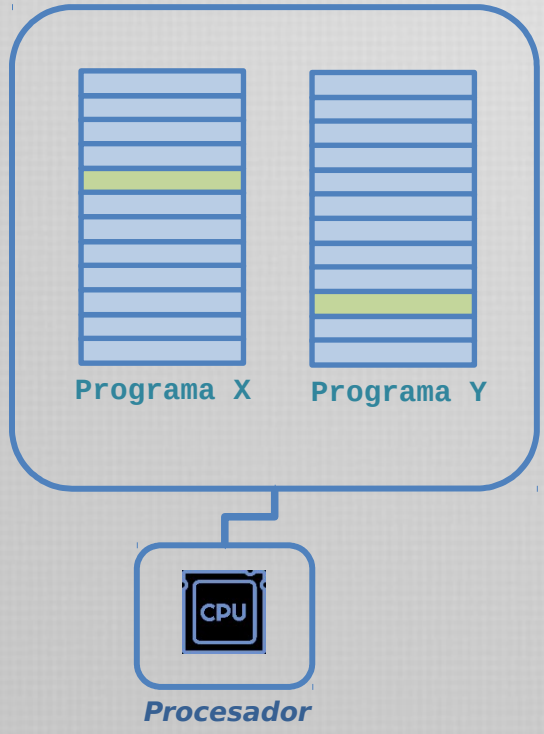
Existe un *instruction pointer para cada uno de ellos.*

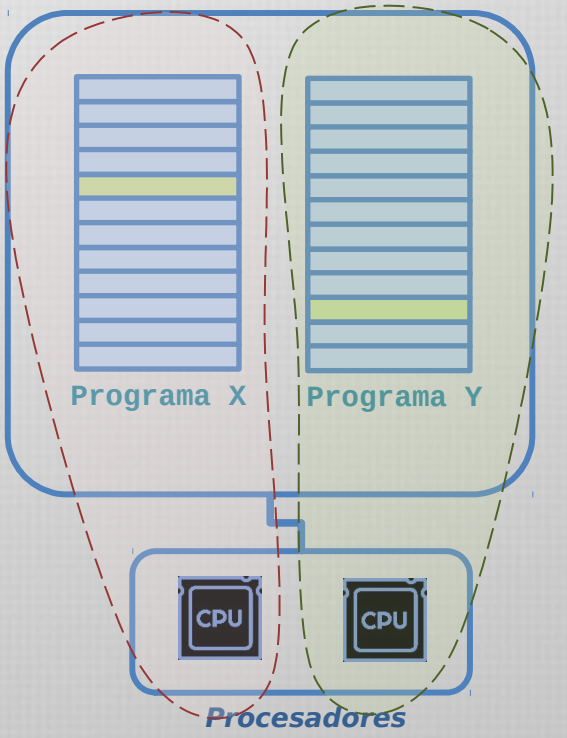
Concurrencia vs Paralelismo

No confundir "concurrente" con "paralelo"

Concurrencia es ejecutar simultáneamente varios programas

Paralelismo es la ejecución simultánea de varios procesadores





Paralelismo ⇒ Concurrencia

Ejemplo – simulación en tiempo real

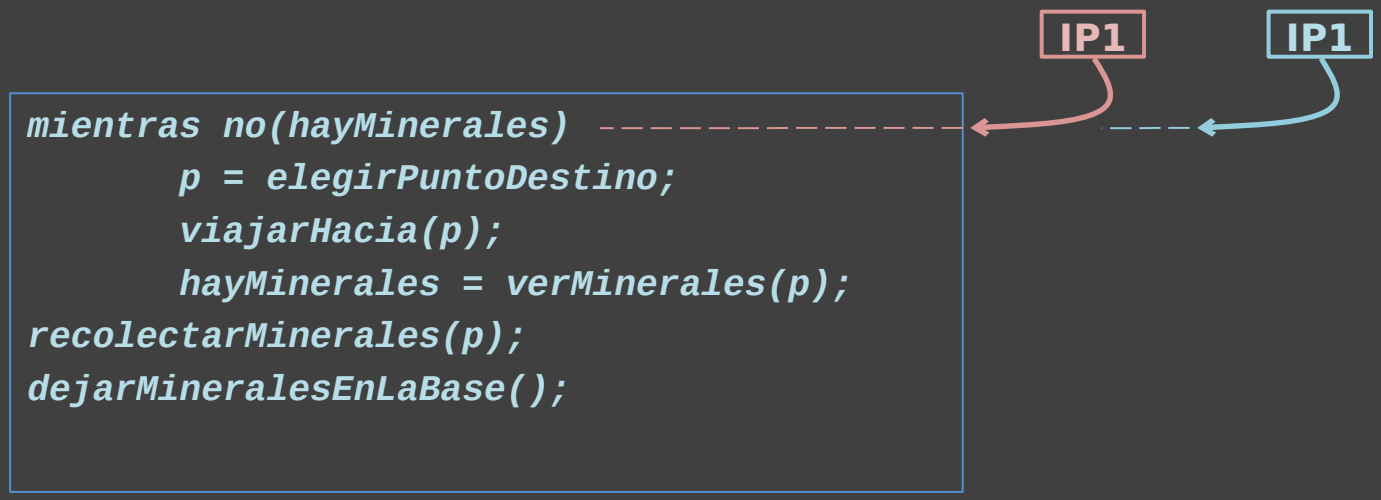
```
RobotExplorador

operación explorar()
    mientras no(hayMinerales)
        p = elegirPuntoDestino;
        viajarHacia(p);
        hayMinerales =
verMinerales(p);
    recolectarMinerales(p);
    dejarMineralesEnLaBase();
```

```
...
r1 = new
RobotExplorador;
r1.explorar();
r2 = new
RobotExplorador;
r2.explorar();
...
```

Una copia única del código.
Cada robot es un hilo de ejecución diferente.

Memoria



Ejemplo – simulación en tiempo real

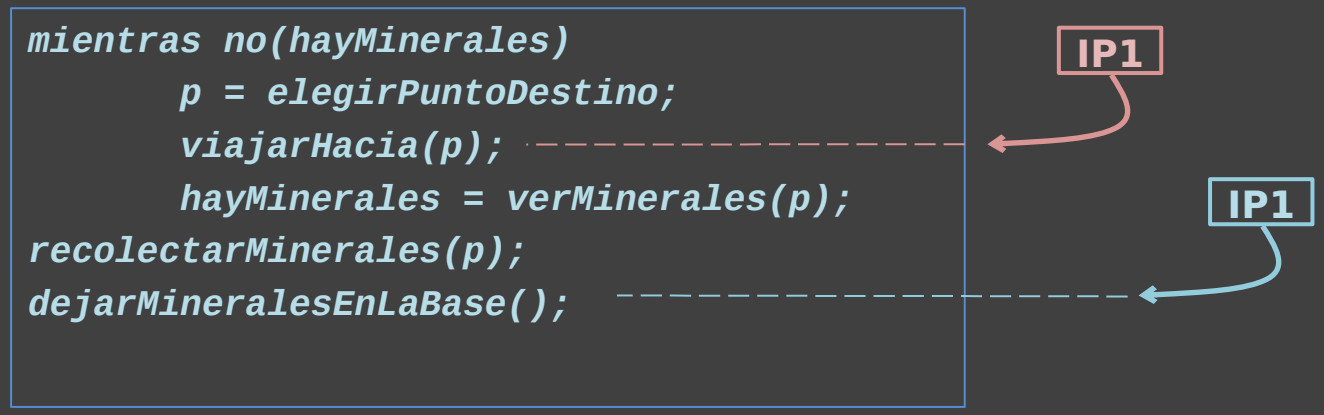
```
RobotExplorador

operación explorar()
    mientras no(hayMinerales)
        p = elegirPuntoDestino;
        viajarHacia(p);
        hayMinerales =
verMinerales(p);
    recolectarMinerales(p);
    dejarMineralesEnLaBase();
```

```
...
r1 = new
RobotExplorador;
r1.explorar();
r2 = new
RobotExplorador;
r2.explorar();
...
```

Una copia única del código.
Cada robot es un hilo de ejecución diferente.

Memoria



Sincronizar es importante

```
f() {
  x = x - 1
}
```

```
g() {
  x = x + 1
}
```

¿cuáles son los posibles resultados de la ejecución concurrente de los dos programas comenzando con x = 0?

```
f() {
  a = x.getValor()
  a = a - 1
  x.setValor(a)
}
```

```
g() {
  a = x.getValor()
  a = a + 1
  x.setValor(a)
}
```

¿cuáles son los posibles resultados de la ejecución concurrente de f y g con x en 0?

Thread-safety implica que el código es seguro de ser utilizado por diferentes hilos de ejecución, manteniendo la consistencia pretendida.

Sincronizar es importante

La sincronización es necesaria para los datos compartidos y asegurar la consistencia (thread-safety).

Existen requerimientos especiales, como
los datos compartidos deben accederse de forma atómica
un proceso puede "reservar" un dato para su uso
las operaciones deben esperar si los datos están en un estado incorrecto

La posibilidad de reservar recursos tambien también trae complicaciones....

```
operacionA()
  reservar(dato1)
  if disponible(dato2) then
        reservar(dato2)
  else
        esperar(dato2)
...
```

```
operacionB()
  reservar(dato2)
  if disponible(dato1) then
        reservar(dato1)
  else
        esperar(dato1)
...
```

operacionA() puede llegar a esperar para reservar un dato que ya está reservado por operaciónB().
Lo mismo puede ocurrir para operacionB() ¡al mismo tiempo!

Esto se denomina deadlock.

Cuidados a tener en cuenta

La concurrencia require especial atención en algunos aspectos:

Seguridad:
los procesos concurrentes pueden manipular equivocadamente datos compartidos.

Ciclo de vida:
un proceso puede “esperar eternamente” si no es manipulado correctamente.

No determinismo:
el mismo programa puede no devolver el mismo resultado al ejecutarse concurrentemente.

Tiempo de ejecución:
la coordinación, el cambio de contexto, y la sincronización, llevan tiempo.

Lenguajes de programación

Los lenguajes de programación deben proveer mecanismos para la ejecución concurrente de código:

Debe permitir especificar procesos concurrentes.

Debe permitir intercambiar información entre procesos.

Debe permitir mantener la consistencia entre procesos (safety)

Existen varias maneras de implementar concurrencia, entre ellas:

Corutinas

operaciones explícitamente indicadas como concurrentes.

Fork / join

clonación de procesos completos.

Cobegin / coend

determinación de bloques de sentencias concurrentes.

Threads en Java

Java permite programación *multithread*

Una operación de un objeto puede ejecutarse en threads diferentes, bajo memoria compartida.

Existen dos formas de obtener objetos con código concurrente

Heredar de la clase *Thread*
Contra: tendrá todos los métodos de la clase Thread.
Pro: probablemente sea la mejor abstracción: un objeto es un thread

Implementar la interfaz *Runnable para un objeto de tipo Thread*
Pro: a veces heredar de Thread es impracticable.
Contra: menos simple.

En los dos casos, la operación concurrente se denomina *run()* y se comienza la ejecución del thread con *start()*.

Threads en Java

```
public class HolaMundo implements Runnable {

    public void run() {
        System.out.println("Hola!");
    }

   // uso de la clase
      public static void main(String args[]) {
        (new Thread(new HolaMundo())).start();
    }

}
```

Implementar la interfaz *Runnable*

```
 public class HolaMundo extends Thread {

     public void run() {
         System.out.println("Hola!");
     }

     // uso de la clase
        public static void main(String args[]) {
         (new HelloThread()).start();
     }

 }
```

Heredar de *Thread*

Clase Thread

La clase *Thread* posee varias operaciones estáticas que permiten manipular threads y conocer el estado de los mismos.

Por ejemplo, la ejecución de un thread puede pausarse por medio de la operación **`Thread.sleep()`**.

```
public class BannerFrases {

public static void main(String args[]) throws InterruptedException {
        String frases[] = {
            "Cuanto cuesta un fin de semana gratis",
            "Solo queda una cerveza, y es de Bart",
            "A la grande le puse Cuca",
            "Lisa, mira detras de ti!"
        };

        for (int i = 0; i < importantInfo.length; i++) {
            Thread.sleep(4000);
            System.out.println(frases[i]);
        }
    }
}
```

Clase Thread

La operación **join()** provoca una espera por la terminación de otro thread

La invocación a la operación **t.join();**
sobre un objeto de t tipo Thread, provoca que
el *thread* actual (el que realiza la sentencia anterior)
espere por la culminación del *thread* t.

```
Thread buscador = new Thread(new BuscadorTrackerTorrent());
buscador.start();
System.out.print("Waiting...");
while (buscador.isAlive()) {
   System.out.print("..");
   // Esperar 5 segundos como máximo
      buscador.join(5000);
      If buscador.isAlive() {
            System.out.print("Basta!");
            buscador.interrupt();
            ...
```

Sincronización de operaciones

Cada objeto tiene asociado un lock, que permite restringir la concurrencia en porciones de código:
sólo un objeto puede acceder al lock al mismo tiempo, los demas deben esperar.

Las operaciones declaradas como *synchronized* impiden la ejecución intercalada entre varios threads.

```
public synchronized void tareaCompleja {
            operacionF();
            operacionG();
            operacionH():
}
```

Cuando un thread X ejecuta **tareaCompleja()**, los demás threads que han invocado esa operación esperan a que X finalice *(pues no tienen el lock)*

Los métodos sincronizados previenen la interferencia entre threads y ayudan a controlar la consistencia.
Sin embargo, aún pueden existir problemas...

Sincronización de sentencias

Podemos sincronizar también bloques de sentencias.
Debemos indicar qué objeto provee el lock (usualmente *this*)

```
public void operacionCompleja() {
   prepararEntorno()
   synchronized(this) {
        lastName = name;
        nameCount++;
   }
   limpiarEntorno();
}
```

Problemas:

```
synchronized(objA) {
  synchronized(objB) {
      hacerAlgo();
  }
}
```

```
synchronized(objB) {
  synchronized(objA) {
     hacerAlgo();
  }
}
```