# Lenguajes de Programación

## Implementación:

### Corrutinas - Iteradores - Excepciones

Ma. Laura Cobo

Universidad Nacional del Sur
Departamento de Ciencias e Ingeniería de la Computación
2018

# Implementación

## Excepciones

Las excepciones tienen dos posibles orígenes:

- **condiciones detectadas por la máquina virtual**.
- **condiciones generadas por la semántica del lenguaje de programación**.

En el caso base, las excepciones del sistema operativo se pueden señalizar directamente a través de interrupciones a nivel hardware o pueden ser señalizadas a software de soporte.

Lenguajes como C permiten al programador acceder directamente a las señalizaciones del sistema operativo. Sólo tiene que activar (enable) una interrupción.

2

# Implementación

## Excepciones

Para las excepciones adicionales, el traductor del lenguaje de programación, debe agregar instrucciones adicionales al código ejecutable.

Es decir, a menos que el harware o sistema operativo provean en chequeo para la excepción; se requerirá simulación en software.

A menudo el costo de estos chequeos en software son importantes, tanto en almacenamiento como en tiempo de ejecución.
Debido a esto, la mayoría de los lenguajes proveen maneras de desactivar el chequeo de excepciones en aquellas partes del programa donde el programados determina que es seguro hacerlo.

3

# Implementación

## Excepciones

Una vez que se señaliza la excepción, la transferencia de control al manejador en el mismo programa implica un salto al comienzo del código del manejador.

La propagación retrocede en la cadena dinámica formada por los punteros de retorno de los registros de activación.

Cada una de las unidades alcanzadas por la propagación debe terminar utilizando una instrucción similar al **return** "especial".

Una vez que se encuentra el manejador apropiado, se lo invoca como a cualquier otra unidad. Sin embargo cuando termina su ejecución, debe también terminar la unidad que lo contiene.

4

# Implementación

## Excepciones

La propagación de excepciones requiere que “retrocedamos” la pila cada vez ¡que el control escapa de la subrutina corriente. El código para de-aloca el registro de activación local debe ser generado por el compilador basado en la relación entre subrutinas.

Otra alternativa, es que haya una rutina de propósito general encargada de “sacar” un registro de activación. Esta rutina seria parte del sistema de run-time.

De manera similar, el manejador mas cercano puede encontrarse a través de código generado por el compilador. Este código mantiene una pila con los handlers activos. O bien inspecciona la tabla de programa-handler generada en compilación.

5

# Implementación

## Eventos

Los eventos se implementan coma llamadas a subritninas "espontaneas"

Dependiendo de la estrategia de implementación, pueden aprovechar el conocimiento de las convenciones de llamado de subrutinas del compilador

Si la llamada es asincrónica, es necesario salvar el conjunto de registros de activacion. Si las llamadas pueden tener lugar, solo en puntos seguros será necerario guardar registros de activación, pero no todos.

# Implementación

## Corrutinas

- **Creación**: deben realizarse las operaciones de inicialización
- **Detach:** luego de independiza (deatch) del programa principal. Esta operación crea un objeto al cual se le puede transferir la ejecución luego y retorna una referencia a su llamador.
- **Transferencia:** guarda el valor concreto del program counter en el objeto creado en el paso previo y reinicia la corrutina indicada como parámetro. El programa principal juega el papel de corrutina default inicial

```
Coroutine check_file_sys
Inicializar
Detach
For all files
        ....
        transfer(us)
```

```
coroutines Update_screen
Inicializar
Detach
Loop
        ....
        tranfer(cfs)
```

```
Begin // main
   us:= new update_screen
   cfs:= new check_file_sys
   transfer(us)
```

7

# Implementación

## Corrutinas

La transferencia entre una unidad y la otra se da a través de la sentencia de transferencia ***resume***. Esta sentencia indica reasumir la ejecución de una activación particular de la corrutina.

Si la corrutina es recursiva y tiene múltiples activaciones, la sentencia no tiene un significado claro.
Es por eso, que la mayoría de los lenguajes que proveen corrutinas, no permiten que exista mas de una activación al mismo tiempo.

Con esta salvedad, la implementación es similar a las llamadas a unidades tradicionales.
Como se permite un solo registro de activación, el mismo es alocado en forma estática al comienzo de la ejecución (como una extensión al segmento de código).

8

# Implementación

## Corrutinas

En el registro de activación se reverva una locación llamada ***resume point***.

Almacena el viejo valor del IP cuando una instrucción *resume* transfiere el control a otra corrutina.
Contrariamente a lo que sucede con el *puntero de retorno* el *resume point* guarda un valor de IP que corresponde a la misma unidad.

De esta manera la ejecución de *resume B* en *A* involucra los siguiente pasos:

1. El valor del CIP se guarda como resume point de A.
2. El valor de IP en el resume point de B es cargado del registro de activación de B y almacenado en CIP (transfiriendo así el control a la correspondiente instrucción de B)

Si se permiten múltiples activaciones se requiere una implementación más compleja

9

# Implementación

## Ejecución Concurrente

Hay dos maneras de implementar un constructor **and**.

**Opción 1:** Si la ejecución puede ser paralela, no se hace ningún tipo de suposición sobre el orden de ejecución. Por lo que tranquilamente se podría ejecutar en secuencia

**Opción 2:** La otra alternativa es utilizar primitivas de ejecución paralela a nivel de sistema operativo (fork)

10

# Implementación

## Ejecución Concurrente

Si la tarea plantea:

**Call** A **and** **Call** B **and** **Call** C;

**Opción 1:**

```
While moreToDo do
    moreToDo := false;
    call A;
    call B;
    call C;
end
```

**Opción 2:**

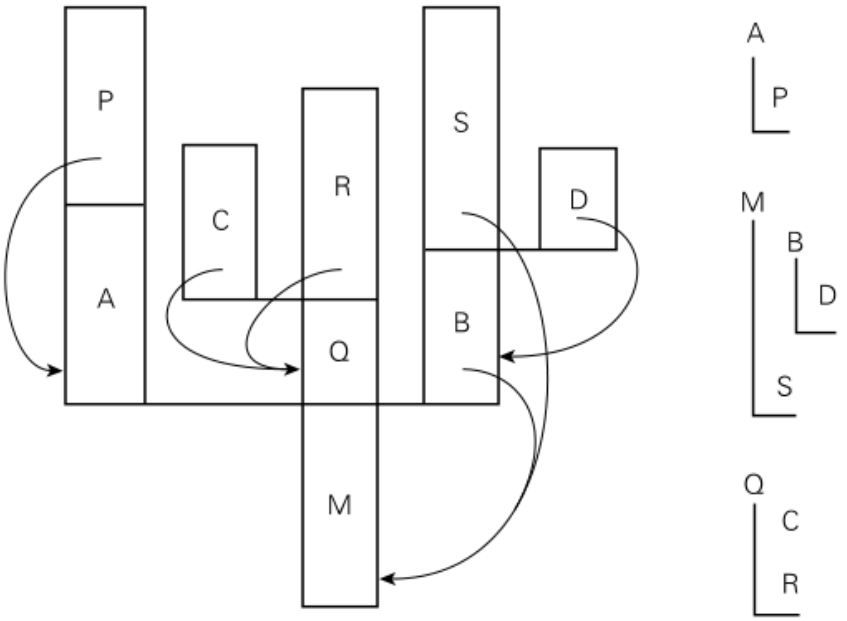
```
fork A;
fork B;
fork C;
wait
```

11

# Implementación

## Stack Allocation

Pila cactus, cada ramificación representa la creación de las corrutinas A,B,C y D.




12

# Implementación

## Iteradores

Teniendo en cuenta la implementación de corrutinas, la implementación de iteradores es casi trivial: una corrutina representa el programa principal; una segunda corrutina se utiliza para representar al iterador.

Si hay iteradores anidados será necesario contar con mas corrutinas.

13

# Implementación

## Iteradores

Para definir un iterador, podemos diseñar tres operaciones:

- **start()**: que inicializa el bloque posicionando el cursor en el primer elemento de la estructura.
- **more()**: que determina si hay un próximo elemento en la estructura.
- **next()**: que se posiciona en el próximo elemento de la estructura si es que hay alguno.

El iterador es como una abstracción del tipo puntero

14

# Implementación

## Iteradores

```
Proc P
 for it=TDA
    se repite para cada elemento de la estrucutura
  for it2. TDA2
}
```

```
Class TDA
    RAISE elem
    brinda elementos con un criterio que queda escondido en la estructura.
    Devuelve elementos hasta que termina
```

Durante la ejecución:

1. se produce la primera invocación al iterador.
2. Se suspende el iterador
3. Reanudación del iterador (vuelve a la unidad, pero debe mantener el estado interno para generar un elemento diferente – mantener comportamiento sensible a la historia)

15

# Implementación

## Iteradores

```
Proc P
 for it=TDA
    se repite para cada elemento de la estrucutura
  for it2. TDA2
}
```

```
Class TDA
    RAISE elem
    brinda elementos con un criterio que queda escondido en la estructura.
    Devuelve elementos hasta que termina
```

Hay que diferenciar la primera activación, que genera el registro de activación, de las demás.

Su uso modifica el comportamiento de pila porque está activa la unidad del iterador y cuando no está activa igualmente se mantiene en el tope de la pila.

Es necesario mantener descriptores de los iteradores en el registro de activación de la unidad que los utiliza

16