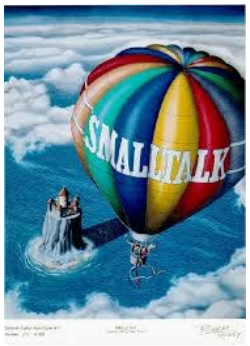
Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue

Programación Concurrente



Procesos en Smalltalk
Semáforos

Smalltalk: le damos lugar a otro proceso

```
[ 1 to: 10 do: [ :i |
  Transcript nextPutAll: i printString, ' '.
  Processor yield ].
Transcript endEntry ] fork.
```

Probar....

```
[ 101 to: 110 do: [ :i |
  Transcript nextPutAll: i  printString, ' '.
  Processor yield ].
Transcript endEntry ] fork
```

En cada bloque mediante la expresión Processor yield, se da la oportunidad a otros procesos con la misma prioridad de ejecutarse.

```
- TRANSCRIPT -
1 101 2 102 3 103 4 104 5 105 6
106 7 107 8 108 9 109 10 110
```

SM-Crear proceso suspendido

```
| pr |
pr := [ 1 to: 10 do: [ :i |
  Transcript show: i  printString ; cr ] ] newProcess.
pr resume
```

Crea el proceso suspendido

No está en la lista de los procesos programados!

Lo añade a la lista del manejador de procesos

SM-Paso de argumentos a un proceso

```
| pr |
pr := [ :max |
  1 to: max do: [ :i |
 Transcript show: i  printString ; cr ] ]newProcessWith:#(20).
pr resume
```

Se pone runnable

Parámetro

También se puede enviar como parámetro un arreglo
NewProcessWith: anArray

SM-Finalización de Proceso

```
| pr |
pr := [ 1 to: 10 do: [ :i |
  Transcript show: i  printString ; cr ] ]fork
pr terminate
```

Finaliza

Una vez que finaliza el proceso **terminate**
no puede recomenzarse nuevamente

SM-Demora

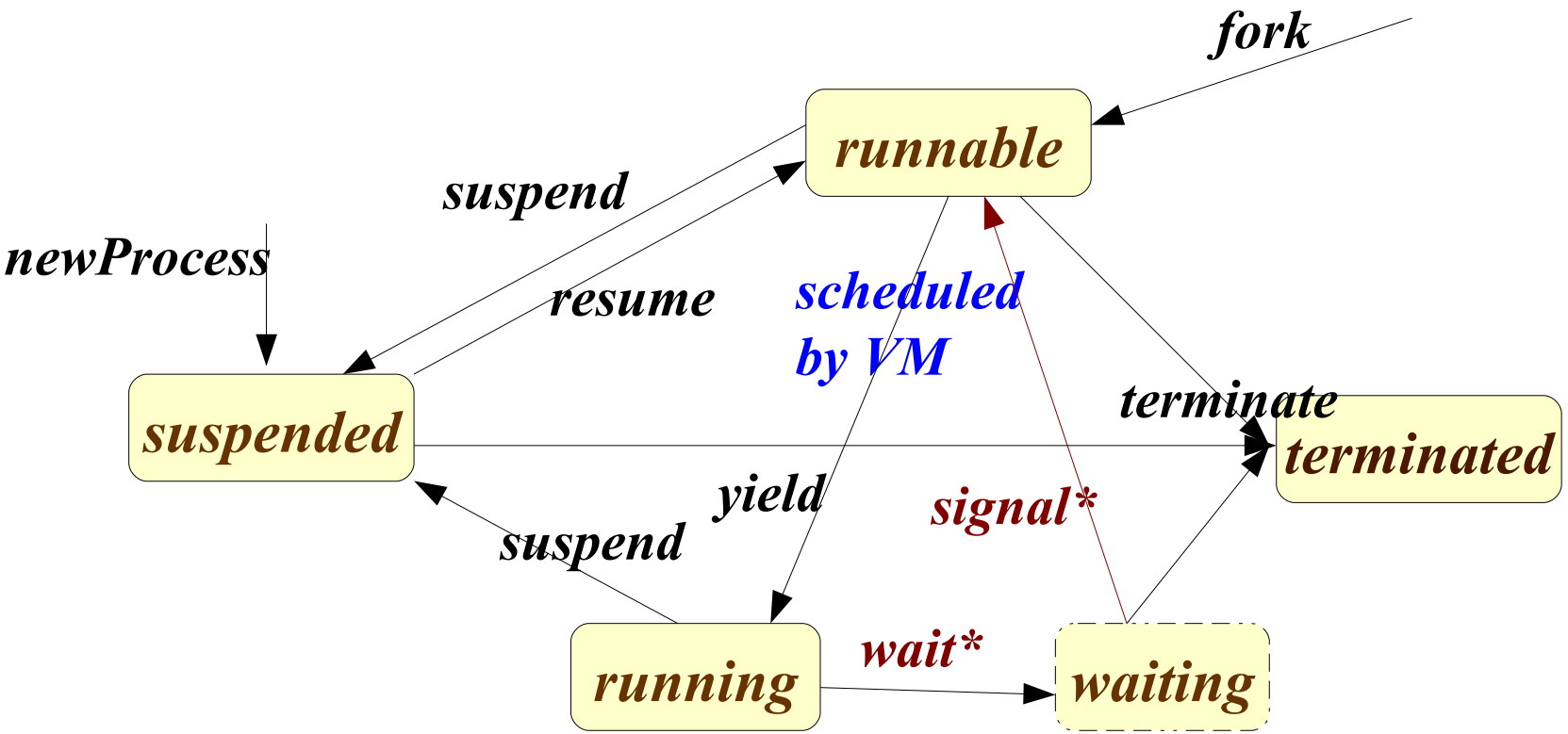
```
delay := Delay forSeconds: 5.
[ 1 to: 10 do: [:i |
  Transcript show: i printString ; cr.
 delay wait ] ] fork
```

Demora

Las pausas pueden ser con **forSeconds:**
Se lo utiliza con wait y con **forMilliseconds:**
a la clase y ejecutado enviando el mensaje **wait**

Estados de un proceso

signal y **wait** se utilizan con un semáforo



SM-procesos

- El manejador de procesos ProcessorScheduler, tiene como única instancia la variable global Processor.
  - Para crear un proceso en estado *runnable* con **fork**
  - Crear un proceso suspendido con **newProcess, newProcess:**
  - Poner un proceso runnable (cuando estaba suspendido) con **resume** .
  - Pasar argumentos a un proceso con **newProcessWith:**
  - Suspender un proceso con **suspend**.
  - Terminar un proceso con **terminate**.
  - Para indicar al manejador que elija otro proceso **Processor yield**
  - Para darle la prioridad a un proceso con **priority:** o **forkAt:**

SM-Manejador de prioridades

- ProcessorScheduler maneja las prioridades.

```
[ 1 to: 10 do: [:i |
  Transcript show: i printString ; cr. ] ] forkAt:
Procesor userBackGroundPriority
```

  - Los procesos de más alta prioridad corren antes de los de baja prioridad.
  - La prioridad es un rango entre 1,,100
  - Las prioridades son para diferentes propósitos:

SM-Prioridades y propósitos

Nro	Squeak	Nombre prioridad	Propósito
100	80	timingPriority	Procesos dependientes en tiempo real
98	70	highIOPriority	Operaciones críticas de I/O
90	60	lowIOPriority	La mayoría de los procesos de I/O
70	50	userInterruptPrioriy	Procesos que requieren servicio inmediatos
50	40	userSchedulingPriority	Procesos con interacciones normales de usuarios
30	30	userBackgroundPriority	Por los procesos que corren en background
10	20	systemBackgroundPriority	Procesos de sistemas que corren en background
1	10	systemRockBottomPriority	La menor prioridad posible

Acceso a una variable compartida

```
| n  p1 p2  d|
d := Delay forMilliseconds: 400.
n:=1.

p1 := [ Transcript show: 'P1 corriendo- n= ',n printString;cr.
    [n < 100000] whileTrue: [
                d wait.
                n := n+1] ].

p2:= [ n :=n+5.
       Transcript show: 'P2 corriendo - n= ',n printString; cr.
       d wait.].

p1 fork.
p2 fork.
```

Define la demora

Variable compartida

Smalltalk-Planificador de procesos

- Sólo hay una instancia.
- Decide cuál proceso se llevará a cabo en un determinado momento (planificador) o sea debe decidir cuál se realizará primero
- Luego se comunica con la maquina virtual para que ésta lleve a cabo efectivamente las tareas representadas por los procesos.
- Debe identificar cuales son los procesos activos en un momento, y suspenderlos o terminarlos, en caso de que haga falta, para luego seleccionar un proceso con tareas pendientes.

ProcessorScheduler

Vbles de instancia: quiescentProcessLists activeProcess'

activePriority, retorna la prioriada actual.

suspendFirstAt: unaPrioridad, retorna suspende el primer proceso que está corriendo con la prioridad pasada como parámetro

activeProcess, retorna el proceso actívo

terminateActive, finaliza el proceso actívo

yield, les da a otros procesos con la misma prioridad la posibilidad de ejecutar

Recordemos...Semáforos

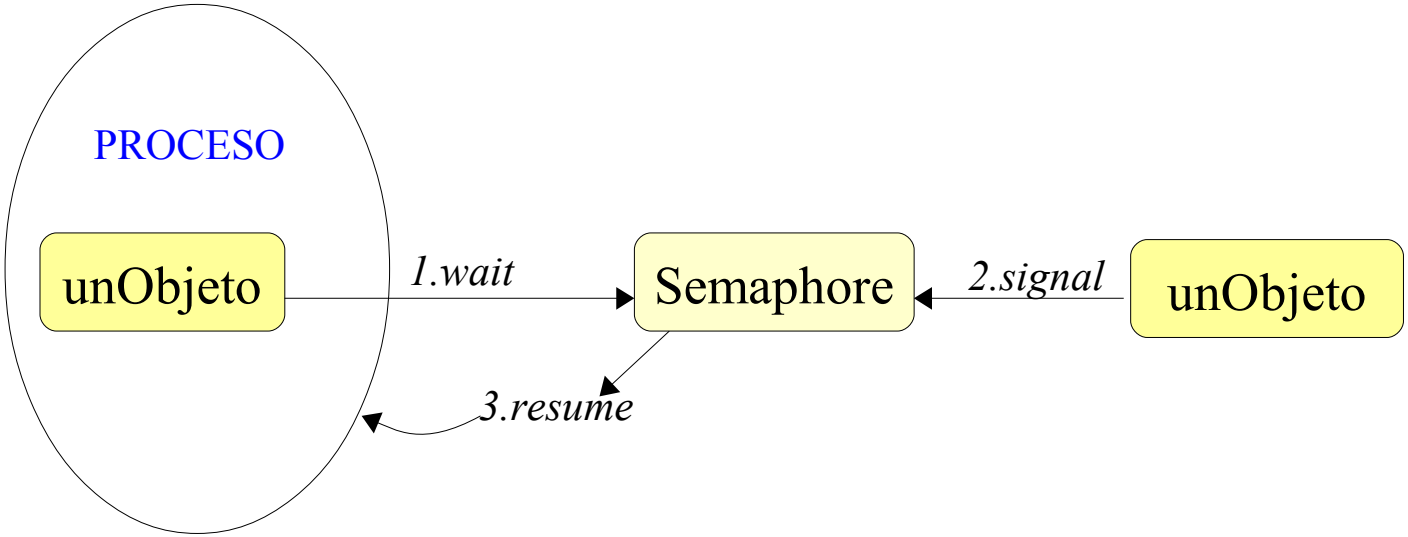
- Proporcionan una forma simple de comunicación sincrónica.
- El semáforo recibe el aviso de un objeto,perteneciente a un proceso, de que necesita que se realice determinada actividad para continuar **(wait -SM)**
- El semáforo espera que del entorno se le envíe una señal **(signal - SM)** para dar por confirmada la realización de la actividad, para finalmente enviarle el mensaje de **Resume** al proceso para que se reanude.

Semáforos en SM

La clase Semaphore, con las operaciones wait y signal

- Lo utilizamos para sincronizar múltiples objetos.
  - El proceso por un evento mandando el mensaje **wait.**
  - Otro proceso envía el mensaje **signal** al semáforo

Utilización de semáforos en Smalltalk



Semaphore	(communication) métodos

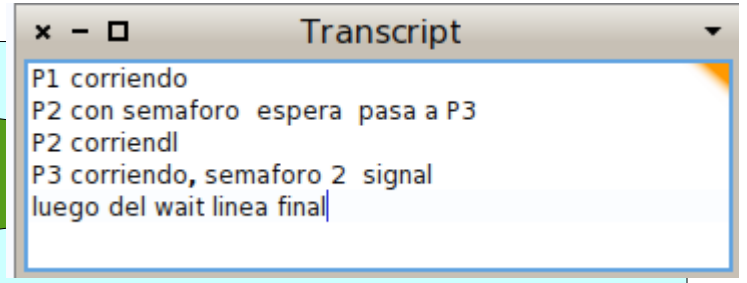
signal, envía un signal a través del receptor. Si uno o más procesos han sidos suspendidos y reciven el signal, permite al primero a proseguir.

wait, el proceso actívo debe recibir un signal a traves del receptor para proseguir. Si no se envía un signa el proceso será suspendido hasta que se envíe uno.

Sincronizar con semáforos

```
|sem|
sem := Semaphore new.

Transcript clear.
[ Transcript show: 'P1 corriendo';cr. ]fork.

[ Transcript show: 'P2 con semaforo en espera, pasa a P3';cr
      sem wait.
     Transcript show: 'P2 corriendol ';cr.]fork.

[ Transcript show: 'P3 corriendo, semáforo signal vuelve a
ejecutar el P2, luego semáforo espera';cr.
      sem signal.
      sem wait.
  Transcript show: 'luego del wait, linea final'.]fork.ork.

  sem signal
```

Semáforo



P1 corriendo

P2 con semaforo en espera, pasa a P3

P2 corriendol P3 corriendo, semáforo signal vuelve a ejecutar el P2, luego semáforo espera

luego del wait, linea final

P1 corriendoP2 con semaforo en espera, pasa a P3P3 corriendo, semáforo signal vuelve a ejecutar el P2, luego semáforo espera

P2 corriendol luego del wait, linea final

Semáforos con exclusión mutua

- El método de instancia que provee exclusión mutua para una sección crítica es **critical:**

(Semaphore new) critical: [un bloque]

- El argumento bloque sólo es ejecutado cuando ningún otro bloque crítico comparte el mismo semáforo

SM-exclusión mutua con semáforos

```
|sem|
sem := Semaphore new.

Transcript clear.
p1:=[Transcript show:'P1 corriendo - n1=',n printString;cr.
    sem critical: [ [n < 100000]
          whileTrue: [
                      d wait.
                     n := n+1]
                     ].
                   ].

p2:= [ n:=n+5.
     Transcript show:'P2 corriendo - n= ',n printString; cr.
     d wait.].

p1 fork.
p2 fork.
```

Semáforo

Sección crítica

Smalltalk - analizar

- Clases
  - Semaphore
  - **Mutex**
    - ***critical: unBloque*** (evalúa unBloque protegido por el receptor)
  - **Monitor**
    - ***critical: unBloque*** (ejecuta unBloque como sección crítica) . Sólo un proceso puede ejecutar esta sección por vez
    - ***wait*** (el proceso corriente es bloqueado y deja el monitor, o sea que el monitor permite a otro proceso ejecutar el código crítico.
    - ***signal*** (despierta un proceso que espera)
    - ***signalAll*** (despierta todos los procesos que esperan)

Semáforos en Java

- Se usan para restringir el número de hilos que pueden acceder a algunos recursos
- Se utiliza conceptualmente para mantener un conjunto de permisos
- Cada vez que se quiere ingresar a una región crítica cada hilo pide permiso al semáforo
- Java provee la clase **Semaphore**
- Los métodos provistos más usados son **acquire()** y **release().**

Semáforos en Java

public void **acquire()** throws InterruptedException

- Adquiere un permiso a este semáforo, bloqueando hasta que este disponible, o que el thread se interrumpa.
- Si no hay ningún permiso disponible, entonces el thread permanece en espera hasta que suceda una de estas dos cosas:
  - Algún otro hilo invoca el método de **release ()** para este semáforo
  - Algún otro thread interrumpe el thread actual.

Semáforos: Algoritmo

```
ALGORITMO bloquear - lock ()
   SI semaforo>0 HACER
         semaforo  ← semaforo -1
   SINO
         bloquea el proceso = duerme
   FIN SI
FIN ALGORITMO
```

```
ALGORITMO liberar - unlock  ()
   SI hay procesoBloqueado  HACER
        despertar proceso = signal
   SINO
        semaforo ← semaforo + 1
   FIN SI
FIN ALGORITMO
```

```
public class Semaforo {  // Java
   private int valor;

   public Semaforo(int v) {valor = v;}
   public acquire(int cant) //libera cant
permisos
   public release(int cant) //libera cant
permisos
   }
```

- Utilización del semáforo en varios procesos
  - valor = 0, algún proceso está en su sección crítica,
  - valor = 1 no hay proceso ejecutándola.

Semáforos en Java

public void **release()**

- Devuelve el permiso, retornandolo al semáforo.
- Aumenta en uno el número de permisos disponibles

Paquete: java.util.concurrent
Class Semaphore

void acquire() //pide un permiso a este semáforo bloqueando hasta que esté disponible

int availablePermits() //retorna el número de permisos disponibles en este semáforo

protected Collection <Thread getQueuedThreads() //retorna la colecciçon de threads esperando

void release() //libera un permiso, lo retorna al semáforo

Semáforo binario casero en Java

SemaforoBin
- contador int
SemaforoBin (int valorInicial)
+**adquirir**() : void
+**liberar**( ): void

Método sincroniado

```
synchronized public void adquirir(){
    while (contador == 0)
        try { wait(); }
        catch (Exception e) {}
        tador--;
    }
```

```
synchronized public void liberar(){
    contador = 1;
    notify(); //despertar a 1 es suficiente
    }
```

Directivas wait() y notify() de la sincronizacion

```
public SemaforoBin (int valorInicial){
    contador = valorInicial;
    }
```

Lectores/escritor

- Un grupo de lectores/escritores quieren tener acceso a un libro.
- Cuando un escritor quiere acceder a un libro éste debe estar desocupado.
    - Lector:
        - Puede haber uno o varios lectores leyendo.
        - Si hay un escritor, entonces el lector deberá esperar a que el escritor acabe para leer
    - Escritor:
        - Si hay un escritor, entonces el escritor que quiere escribir debe esperar a que no haya nadie leyendo.

Utilizar:
-Semáforo,
-Lector,
-Escritor