Lección 12: Ejemplos de programación con monitores

- El problema del semáforo equitativo
- El problema del temporizador
- El problema del barbero dormilón

El problema del semáforo equitativo

- Era correcto
- Comportamiento **no equitativo**: el proceso que recibe un *signal* no tiene garantizada la inmediata reanudación de su ejecución
  - el *signal* puede ser "robado"

```
Monitor semaforo
  Vars s: Ent:= 0
       esPos: cond  --señala s>0
                    --IM::Sem: s>=0
  Accion wait
  Principio
     Mq (s=0) wait(esPos) FMq
     s := s-1
  Fin
  Accion send
  Principio
     s := s+1
     signal(esPos)
  Fin
FMonitor
```

El problema del semáforo equitativo

```
Monitor semaforo   --IM::Sem: s>=0
  Vars s: Ent:= 0
       esPos: cond --señala s>0 cuando
                   --esPos no está vacía
  Accion wait
  Principio
     Sel
       s>0: s:=s-1
       s=0: wait(esPos)
     FSel
  Fin

FMonitor
```

```
...
Accion send
Principio
  Sel
    empty(esPos):  s:=s+1
    ¬empty(esPos): signal(esPos)
  FSel
Fin
...
```

El problema del temporizador

- Diseñar un monitor con las operaciones *retardar* y *tick* que se especifican a continuación

```
IM::Rel: t>=0 ∧
    t es monótono de 1 en 1
```

```
Monitor temporizador
  Vars t: Ent:= 0

  Accion delay(E int: Ent)
  --Pre:  Rel∧int>=0∧t=T
  --Post: Rel∧t>=T+int

  Accion tick
  --Pre:  Rel∧t=T
  --Post: Rel∧t=T+1
FMonitor
```

El problema del temporizador

```
Monitor temporizador
  Vars t: Ent:= 0
       comprobar: cond  --¡Un nuevo tick!

  Accion delay(E int: Ent)
    Vars horaDeDespertar: Ent
  Principio
    horaDeDespertar := t+int
    Mq t<horaDeDespertar wait(comprobar) FMq
  Fin
  Accion tick
  Principio
    t := t+1
    signal_all(comprobar)
  Fin
FMonitor
```

IM::Rel: t>=0 ∧
t es monótono de 1 en 1

El problema del temporizador

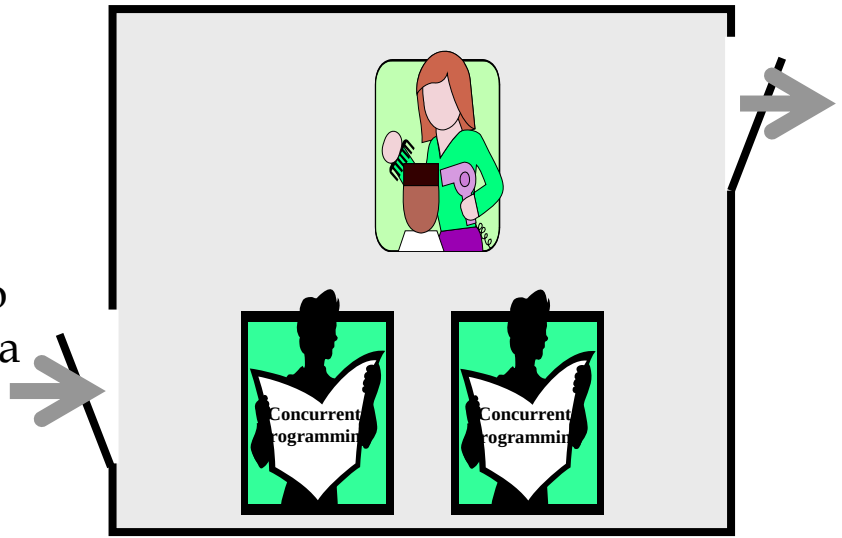
Una solución con esperas con prioridad

```
Monitor temporizador
  Vars t: Ent:= 0
       comprobar: cond  --minrank(comprobar)<=t
  Accion delay(E int: Ent)
    Vars horaDeDespertar: Ent
  Principio
    horaDeDespertar := t+int
    Si t<horaDeDespertar wait(comprobar,horaDeDespertar) FSi
  Fin
  Accion tick
  Principio
    t := t+1
    Mq ¬empty(comprobar)∧minrank(comprobar)<=t
        signal(comprobar)
    FMq
  Fin
FMonitor
```

```
IM::Rel: t>=0 ∧
    t es monótono de 1 en 1
```

El problema del barbero dormilón

- Cuando no hay clientes, el barbero duerme en un sillón de trabajo
- Si entonces llega un cliente, éste despierta al barbero, se sienta en el sillón y se duerme mientras el barbero le corta el pelo
- Si llega un cliente cuando el barbero está trabajando, se sienta en una silla de espera y duerme hasta que llega su turno y, entonces, el barbero le despierta
- Cuando ha terminado, el barbero le abre la puerta y cuando se ha ido, va a por el siguiente cliente



Análisis y diseño del programa

- **Elementos principales** del programa concurrente:
  - Barbero: proceso servidor
  - Clientes: procesos clientes
  - Barbería: monitor para sincronizar el *barbero* y los *clientes*
- Variables para la **sincronización**:
  - *clientesEnSilla*: #clientes han pasado por el sillón del barbero
  - *clientesSalido*: #clientes que han sido servidos y cobrados
  - *barberoDisponible*: #veces en que el barbero ha estado en disposición de atender a un nuevo cliente
  - *barberoOcupado*: #veces en que el barbero ha estado trabajando
  - *barberoTerminado*: #veces en que el barbero ha acabado un servicio

Requisitos de sincronización

- **C1**: los clientes reciben el servicio antes de pagarlo y el barbero debe estar disponible antes de atender a un cliente y sólo después de servirle puede cobrarle:

```
          (clientesEnSilla >= clientesSalido) ∧
(barberoDisponible >= barberoOcupado >= barberoTerminado )
```

- **C2**: un cliente no puede sentarse en el sillón del cliente si el barbero está atendiendo a otro cliente y el barbero no comienza un nuevo servicio si no tiene un cliente en el sillón del cliente:

```
    (clientesEnSilla <= barberoDisponible) ∧
      (barberoOcupado <= clientesEnSilla)
```

Requisitos de sincronización

- **C3**: para que el n-ésimo cliente pague será preciso que antes el barbero haya concluido el servicio y esté dispuesto a cobrarle:

```
clientesSalido <= barberoTerminado
```

- Invariante:

$$\mathbf{Barb:\ C1 \wedge C2 \wedge C3}$$

Una primera solución

```
Monitor barbería
  Vars clientesEnSilla:    Ent := 0
       clientesSalido:     Ent := 0
       barberoDisponible:  Ent := 0
       barberoOcupado:     Ent := 0
       barberoTerminado:   Ent := 0

  Accion cortemeElPelo --llamada por los clientes
  Principio
    <await clientesEnSilla < barberoDisponible>
    clientesEnSilla := clientesEnSilla+ 1
    <await clientesSalido < barberoTerminado>
    clientesSalido := clientesSalido + 1
  Fin
  ...
FMonitor
```

IM: Barb: C1 ∧ C2 ∧ C3

Una primera solución

IM: Barb: C1 ∧ C2 ∧ C3

```
...
Accion siguientePorFavor --llamada por el barbero
Principio
  barberoDisponible := barberoDisponible + 1
  <await barberoOcupado < clientesEnSilla >
  barberoOcupado := barberoOcupado + 1
Fin

Accion servicioConcluido --llamada por el barbero
Principio
  barberoTerminado := barberoTerminado + 1
  <await barberoTerminado = clientesSalido >
Fin
...
```

Una solución con variables condición

```
IM: Barb': 0<=barbero<=1 ∧ 0<=silla<=1 ∧
           0<=abierta<=1
```

```
Monitor barbería
  Vars barbero: Ent := 0
       silla: Ent := 0
       abierta: Ent := 0
       barberoDisponible: cond --cuando barbero>0
       sillaOcupada: cond      --cuando silla>0
       puertaAbierta: cond     --cuando abierta>0
       clienteSale: cond       --cuando abierta=0

  ...
FMonitor
```

```
barbero: barberoDisponible-clientesEnSilla
silla:   clientesEnSilla-barberoOcupado
abierta: barberoTerminado-clientesSalido
```

Una solución con variables condición

```
IM: Barb': 0<=barbero<=1 ∧ 0<=silla<=1 ∧
           0<=abierta<=1
```

```
Accion cortemeElPelo --llamada por los clientes
Principio
  Mq (barbero=0) wait(barberoDisponible) FMq
  barbero := barbero-1
  silla := silla+1
  signal(sillaOcupada)
  Mq (abierta=0) wait(puertaAbierta) FMq
  abierta := abierta-1
  signal(clienteSale)
Fin
```

```
barbero: barberoDisponible-clientesEnSilla
silla:   clientesEnSilla-barberoOcupado
abierta: barberoTerminado-clientesSalido
```

Una solución con variables condición

```
Accion siguientePorFavor --llamada por el barbero
Principio
  barbero := barbero+1
  signal(barberoDisponible)
  Mq (silla=0) wait(sillaOcupada) FMq
  silla := silla-1
Fin

Accion servicioConcluido --llamada por el barbero
Principio
  abierta := abierta+1
  signal(puertaAbierta)
  Mq (abierta>0) wait(clienteSale) FMq
Fin
```

```
IM: Barb': 0<=barbero<=1 ∧ 0<=silla<=1 ∧
           0<=abierta<=1
```

```
barbero: barberoDisponible-clientesEnSilla
silla:   clientesEnSilla-barberoOcupado
abierta: barberoTerminado-clientesSalido
```

Una solución con variables condición

```
barbero::
    Mq TRUE
       barberia.siguientePorFavor
       barberia.servicioConcluido
    FMq
```

```
cliente::
    barberia.cortemeElPelo
```

Una solución con variables condición

```
Monitor barbería
  Vars barberoDisp: Ent := 0 --cuando barberoDisp: 0,1
       sillaOcup: Ent := 0
       puertaAb: Ent := 0
       barberoDisponible: cond --cuando barberoDisp>0
       sillaOcupada: cond      --cuando sillaOcup>0
       puertaAbierta: cond     --cuando puertaAb>0
       clienteSale: cond       --cuando puertaAb=0


  ...
FMonitor
```

Una solución con variables condición

```
Accion cortemeElPelo --llamada por los clientes
Principio
  Mq barberoDisp=0
     wait(barberoDisponible)
  FMq
  barberoDisp := 0
  sillaOcup := 1
  signal(sillaOcupada)
  Mq puertaAb=0
     wait(puertaAbierta)
  FMq
  puertaAb := 0
  signal(clienteSale)
Fin
```

Una solución con variables condición

```
Accion siguientePorFavor --llamada por el barbero
Principio
  barbDisp := 1
  signal(barberoDisponible)
  Mq sillaOcup=0
     wait(sillaOcupada)
  FMq
  sillaOcup := 1
Fin
Accion servicioConcluido --llamada por el barbero
Principio
  puertaAb := 1
  signal(puertaAbierta)
  Mq puertaAb=1
     wait(clienteSale)
  FMq
Fin
```