Técnicas de implementación con objetos protegidos

Lecturas:
Transparencias y apuntes de la asignatura

Manuel Carro

Universidad Politécnica de Madrid

Este texto se distribuye bajo los términos de la *Creative Commons License*

A grandes rasgos

- Dos pasos principales
  (tras procesos y recursos y especificación):
  1. Implementación seguridad
  2. Implementación vivacidad

Seguridad: asegurar resultados correctos
(debidos a mala sincronización)

Vivacidad: modificar anterior para que procesos se *comporten* bien
(no debe comprometer seguridad)

Técnicas vistas para seguridad

- CPRE no depende de parámetros entrada
  $\rightarrow$ directamente a **when**
- CPRE dependiente de parámetros:
  - Desdoblamiento de operaciones con nuevas *entries* "a mano"
  - Crear familia de *entries*
- Más posibilidades:
  - Forzar atención en orden de llegada (*uno a uno*)
  - Encolado en *entries* internas (generalización de la anterior)
  - Bajar al nivel de tareas individuales (permite desbloqueo explícito de tareas)

Atención por orden de bloqueo

- Aplicable a veces, pero en general no adecuada
- Espera a que se atienda completamente una llamada antes de dar paso a otra en la misma operación
- Idea: pasar parámetros de llamada a estado
- Usar variable para señalar bloqueo

```
entry OpX(Param)
when not Bloqueada is
begin
  Aux := Param;
  Bloqueada := True;
  requeue OpX_Aplazada;
end OpX;
```

```
entry OpX_Aplazada(Param)
when Cond(Aux, Estado) is
begin
  <Atender peticion>
  Bloqueada := False;
end OpX_Aplazada;
```

- ¿Qué ocurre si when not Bloq $\Longrightarrow$ when True?

Atención por orden de bloqueo

Características

- **Registro**: guarda **una** instancia de **Parámetros**
- No importa el tipo de los parámetros
- Máximo una tarea bloqueada en cada *entry*
- Debe esperarse a que se atienda esa petición
- Comprobación completa de condiciones en *entry* interna
- Posibles problemas de vivacidad (estudiar requisitos del sistema)

Ejemplo: invalidación de un dato

```
protected type Tipo_Espera is
  ...
  DatoInt: TipoDato;
  Bloq: Boolean := False;
end TipoEspera;

entry Espera
      (D: TipoDato)
  when not Bloq is
begin
  Bloq:= True;
  DatoInt:= D;
  requeue Espera_Apl;
end Espera;
```

```
entry Espera_Apl
        (D: TipoDato)
  when HayDato and
         DatoInt = Dato is
begin
  Bloq:= False;
  HayDato := False;
end Espera_Apl;
```

- ¿Es correcto?
- ¿Es adecuado (vivacidad)?
- ¿Serviría para implementar un servicio estilo *cola de espera*?

Doble bloqueo

- Bloqueo en orden de llegada: caso particular de doble bloqueo
  - Primero en *entry* pública: usa sólo estado (CPRE *debilitada* — puede ser True)
  - Después en *entry* privada: estado aumentado con argumentos (debe ser equivalente a CPRE completa)
- CPRE debilitada: evita paso a *entry* privada (mejor rendimiento en algunos casos)
- Guarda interna debe expresar CPRE completa

Familias de *entries* internas

CPRE: C(r, p)
Op(r, p)

- ¿Cómo bloquear más de una llamada?
- Familia de *entries* interna

```
entry Op(P: T1)
when True is
begin
  requeue Op_Apl(Proyecta(P));
end Op;
```

```
entry Op_Apl(for I in T2)
             (P: T1)
when C1(R, I) is ...
```

Proyecta : $T1 \mapsto T2$ debe cumplir

$$C(R,P) \Leftrightarrow C_1(R, Proyecta(P))$$

Fácil si tenemos $Proyecta^{-1}$

Bloqueo interno más potente

- Desviar a familia de *entries* interna:
  - A veces parámetros no fáciles de linealizar
  - A veces rango parámetros excesivo
- A menudo número tareas menor que cantidad *entries*
- Idea: descender al nivel de tarea
  - Identificar cada tarea llamante y asociar sus argumentos con el identificador
  - Recorrer estado(s) almacenado(s) y decidir qué tareas rearrancar
- El identificador debe poder ser índice de una familia de *entries*

Identificación global de tareas

- Dando identificador directamente al crearla:

```
task type Cliente(Id: Tipo_PID);
...
T(I) := new Cliente(Id);
```

- Utilizando el paquete Ada.Task_Identification. Cada tarea tiene su propio Task_ID.
- Se utiliza el identificador de la tarea en las llamadas que lo necesiten.

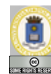
Identificación local de tareas

- Haciendo que la tarea adquiera identificador de un conjunto:

```
task type Cliente;
  Id: Tipo_PID;
begin
  ...
  Id := Cnj_Id.Nuevo_PID(...);
```

  y utilizar Id en las llamadas que lo necesiten.
- Generación de identificadores antes de requeue dentro del propio recurso (ver siguiente ejemplo).
- Los identificadores pueden *devolverse* cuando ya no sean necesarios.

Reclamaciones



- Reclamaciones a un comité (por correo)
- Resultado: aceptada, desestimada, denegada
- Recurso: ventanilla de comunicación de resultados

```
loop -- Comité
   <Elegir reclamación a revisar, emitir dictamen>
   Ventanilla.Opinar(Persona, Dictamen);
end loop;
```

```
task type Tipo_Persona (Persona : Tipo_DNI);
...
loop -- Reclamante
  <Enviar reclamación al comité>
  Ventanilla.Reclamar(Persona, Resultado);
  <Actuar conforme a Opinión>
exit when Resultado /= Desestimado;
end loop;
```

Recurso OMIC

(Oficina Mundial de Información al Consumidor)

- Recurso notifica sobre estado de apelación
- Comité puede estar pensando

TIPO: *Tipo_OMIC = (reclamante:* $\mathbb{N}\times$ *opinion: Tipo_Estado)*
Tipo_Estado = denegado | aceptado | desestimado | pensando

CPRE: *cierto*
Opinar(com, dni, res)
POST: $com^{sal} = (dni, res)$

CPRE: $com = (dni, op) \wedge op \neq pensando$
Reclamar(com, dni, res)
POST: $com^{ent} = (_, res^{sal}) \wedge com^{sal} = (_, pensando)$

Implementación de *Reclamar*

Consideraciones y estado del recurso

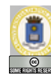
- Número personas grande $\Longrightarrow$ familia *entries* inadecuada
- Atención *uno a uno* inadecuada (**¿Por qué**)
- Si relativamente pocos apelan: asignación dinámica de identificadores
- Utilizar identificador (NR, número de registro) para guardar datos y bloquear

```
type Tabla_Casos is ...  -- Tiene tamaño máximo

private
   Casos: Tabla_Casos;  -- NR |--> DNI
   Opinion: Tipo_Estado := Pensando;
   Reclamante: Tipo_DNI;
   ...
```


Implementación de *Reclamar*

Operacion



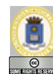
```
entry Reclamar(Dni: in Tipo_DNI;
               Res: out Tipo_Estado)
when True is  -- Dejamos anotar petición
    NR: Tipo_NR;
begin
    NR := Buscar_Libre(Casos);   -- Un número libre
    Asignar(Casos, NR, Dni);     -- Lo asociamos con Dni
    requeue Reclamar_Apl(NR);
end Reclamar;
```

Implementación de *Reclamar* (Cont.)

- Casos implementa Tipo_NR ↦ Tipo_DNI
- La función $Proyecta^{-1}$, tal y como queríamos (ver transparencias anteriores)
- Sólo guarda las personas que han pedido apelación

```
entry Reclamar_Apl (for NR in Tipo_NR)  -- ¡Pocos!
                   (Dni: in Tipo_DNI;
                    Res: out Tipo_Estado)
when Opinion /= Pensando and
     Recuperar(Casos, NR) = Reclamante is
begin
   Res := Opinion;
   Opinion := Pensando;
   Borrar(Casos, NR);  -- Para reusar el NR
end Reclamar_Apl;
```


Opinión y cuestiones



- Opinar: sólo notifica del resultado de una deliberación

```
entry Opinar(Dni : in Tipo_DNI;
             Res : in Tipo_Estado)
when True is
begin
   Reclamante := Dni;
   Opinion := Res;
end Opinar;
```

- Despertar automático: todas las *entries* inspeccionan Reclamante (y Opinion)
- Además: sólo una *entry* (un consumidor) puede despertarse
- ¿Podría un resultado evaluado por el comité no ser nunca recibido por un reclamante? En caso afirmativo, ¿cómo evitarlo?

Cuenta compartida

Uso de desbloqueos explícitos

- Dinero en cuenta; padres ingresan, hijos retiran

C-TADSOL Cuenta
OPERACIONES
ACCIÓN Ingreso: *Tipo_Cuenta[es]* × *Tipo_Saldo[e]*
ACCIÓN Reintegro: *Tipo_Cuenta[es]* × *Tipo_Saldo[e]*

SEMÁNTICA
DOMINIO:
TIPO: *Tipo_Cuenta* = *Tipo_Saldo*
INICIAL(b): b = 0

CPRE: Cierto
Ingreso (b, s)
POST: $b^{sal} = b^{ent} + s$

CPRE: $b \geq s$
Reintegro (b, s)
POST: $b^{sal} = b^{ent} - s$

Estrategias de implementación

- Familia de *entries* (una por cada posible saldo)
  - Posible (si saldo máximo no muy grande): `account_ent_fam.adb`
  - Aparte de indeterminismo entre *entries*: preferencia **intrínseca** por peticiones más bajas → ¿prioridad no planteada?
- Atención por orden de bloqueo
  - Más justa (para este caso): tarea espera a ser servida (`account.adb`)
- Planificación más sofisticada: política decide a nivel de tarea
  - Guardar datos en estado
  - Decidir qué tarea rearrancar

Cuenta compartida

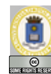
- Estado aumentado: saldo, tareas bloqueada, peticiones tareas
- Dejar datos en estado:

```
entry Reintegro(Cant: ...; I: Tipo_PID)
when True is begin  -- I: externo o adquirido aquí
  A_Reintegrar(I) := (True, Cant);
  requeue Reintegro_Apl(I);
end Reintegro;
```

- *Entry* interna: bloquear en familia de *entries* con información de vivacidad:

```
entry Reintegro_Apl(for I in Tipo_PID)(...)
when A_Reintegrar(I).Saldo =< Saldo and -- Seguridad
     Mejor_Peticion(I)                  -- Vivacidad
is ...
```


Cuenta compartida y vivacidad



- Mejor_Peticion(I) implementa política rearranque
- Determinista si arranca sólo un reintegro en cada momento
- Podemos afinar lo que queramos:
  - Petición menor antes (inanición)
  - Petición mayor antes (también inanición — ¡encontrar caso!)
  - Mayor petición entre las menores que el saldo (también inanición — ¡encontrar caso!)
  - Posibilidad: implementar turno rotativo entre peticiones
- Puntos anteriores independientes de método de implementación (debemos poder implementar **cualquier** política de atención)
- ¿Por qué el componente booleano de Llamadas? ¿Por qué no se usa en Reintegro_Apl?
- Veremos cómo localizar problemas de vivacidad

Una crítica y una posibilidad

- ¿Cómo es de caro Mejor_Peticion(I)? (habrá una llamada por *entry* a considerar)
  - Mejor_Peticion() puede recorrer todas las llamadas bloqueadas — una vez por llamada bloqueada: $O(n^2)$
  - Mejor_Peticion() no toma decisiones globales
- Desbloqueo explícito: ingreso puede desbloquear reintegro
  - → Comprobar explícitamente tras hacer ingreso
  - → Bloquear ingresos en variable separada
  - → Calcular valor variable explícitamente
- Técnica de desbloqueos explícitos

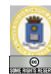
Ejemplo de desbloqueo explícito

(account_expl.adb)

```
entry Reintegro (Id   : in Tipo_Pid;
                 Cant: in Tipo_Peticion)
when True is begin
   Cant_Pedida(Id) := Cant;
   Bloqueado(Id)   := True;
   Computar_Desbloqueo;
   requeue Reintegro_Apl (Id);
end Reintegro;

entry Reintegro_Apl (for Pid in Tipo_Pid)
      (Id    : in Tipo_Pid;
       Cant  : in Tipo_Peticion)
when not Bloqueado(Pid) is begin
   Saldo := Saldo - Cant;
   Computar_Desbloqueo;           -- ¡Cascada!
end Reintegro_Apl;
```


Desbloqueos explícitos



- Detectar operaciones a desbloquear tras cada operación
- ¡Posibilidad de inanición si preferencia estática!
- Desbloqueo de varias operaciones: mismo problema que con varias guardas abiertas
- Dar paso sólo a una tarea:
  - A lo sumo un Bloqueado(I) falso en cada momento
  - Usar una variable que da turno a la siguiente entry
- Operación **A** permite rearrancar otras (**B** y **C**)
  - **A** puede desbloquear sólo a una (ej., **B**)
  - **B** debe considerar desbloquear **C** — aunque su código no cambie estado en esa dirección: **Despertar en cascada**
  - Ej., Reintegros desbloquean otros Reintegros

Desbloqueos explícitos (Cont.)

- Esquema: comprobar condiciones tras cada operación
- Comprobación agrupable en procedimiento separado que revisa todas las condiciones (Computar_Desbloqueo)
- A menudo: no todas las combinaciones tienen sentido (p.ej., Poner en *buffer* no puede despertar otra Poner)
- Puede especializarse código desbloqueo en cada operación (Poner sólo comprueba condición de Tomar)
- ¿Cuándo afinar hasta desbloqueo explícito? ¿Qué operaciones considerar en cada desbloqueo?
- **Análisis de vivacidad**

Estudio de seguridad y vivacidad

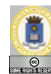
- Tablas seguridad / tablas vivacidad
- Independientes del lenguaje

Seguridad: recoger condiciones bloqueo
- Dependientes / independientes datos entrada
- Generar tabla de bloqueos

Vivacidad: estudiar desbloqueos
- Qué operaciones pueden / deben despertar a otras
- Operaciones que pueden no activarse nunca
- Generar tabla de POST ampliadas: incluyen más información

- Problemas vivacidad (ej., inanición):
  - A veces dependientes de características implementación
  - Intrínsecos al problema (cuenta compartida)

Tablas seguridad (ind. parám.)



- Nombre operación, precondición informal, dependencia parámetros, codificación

Op	C-PRE inf	¿Dep. parám.?	C-PRE cod.
Nombre	...	si/no	...

- **Ejemplo:** *buffer* acotado
- n número elementos en el *buffer*

Op.	C-PRE inf.	dep.	C-PRE cod.
Poner (D: in TDato)	No lleno	no	$n < MAX$
Tomar (D : out TDato)	No vacío	no	$n > 0$

Tablas seguridad (dep. parám.)

- Sólo un poco más complejas
- CPRE Φ: descomponer en diversas CPRE
- Proyección sobre parámetros a:

$$\Phi \quad [a \mapsto x_1] \quad \vdots \quad \Phi \quad [a \mapsto x_n] \qquad \Phi = \bigvee_{x \in D(a)} \Phi[a \mapsto x]$$

- **Ej:** Φ = (HayDato ∧ ((Dato mod 2 = 0) = (Req = Even)))
  Req ∈ { Even, Odd }
- $\Phi[Req \mapsto Even]$ = (HayDato ∧ (Dato mod 2 = 0))
- $\Phi[Req \mapsto Odd]$ = (HayDato ∧ (Dato mod 2 ≠ 0))

Tablas seguridad (dep. parám. — Cont.)

Operación	**C-PRE inf.**	**dep.**	**C-PRE cod.**
Op (...)	CPRE	**sí**	$\Phi[a \mapsto x_1]$
			$\vdots$
			$\Phi[a \mapsto x_n]$

- *Buffer* par/impar

Operación	**C-PRE inf.**	**dep.**	**C-PRE cod.**
Poner(D: in TDato)	No lleno	no	$\neg Hay$
Tomar(P: in TPet; D: out TDato)	Hay dato de tipo P	**sí**	$Hay \wedge (Dato\ mod\ 2 = 0)$
			$Hay \wedge (Dato\ mod\ 2 = 1)$

Estudio vivacidad

- Al acabar una operación, ¿qué otras pueden empezar?

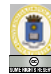
→ Posibilidad de inanición

- ¿Qué es verdad al acabar una operación?
  - POST, invariante, partes de la CPRE, modo de uso ... → POST$^+$
- Hace más explícito estado tras operación
- ¿Puede *arrancar* Op$_j$ al terminar Op$_i$?

$$POST_i^+ \wedge c_{i,j} \rightarrow CPRE_j$$

- $c_{i,j}$ la más debil que cumple lo anterior
- Si $c_{i,j} \equiv falso$, Op$_j$ no puede ejecutarse tras Op$_i$

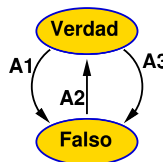
Ejemplo tabla desbloqueos



POST	**Poner(b,d)** $\neg EstaLlena(b)$	**Tomar(b,d)** $\neg EstaVacia(b)$
Poner(b,d) $b^{sal} = Insertar(b^{ent}, d)$	$\neg EstaLlena(b)$ (*no aplicable*)	cierto
Tomar(b,d) $b^{sal} = Borrar(b^{ent}) \wedge$ $d^{sal} = Primero(b^{ent})$	cierto	$\neg EstaVacia(b)$ (*no aplicable*)

Grafo de transiciones

- Estados del recurso en nodos
- Transformación entre estados: arcos etiquetados con la operación que transforma

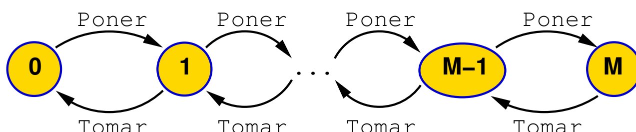


- Observar historia de transiciones (vivacidad relacionada con historia)
- Toda transición ($\equiv$ llamada a operación) debería tener posibilidades
- Estado con varias transiciones simultáneamente activas: posible inanición (¿cuál tomar?)
- Conjunto de estados mutuamente alcanzables que excluyen determinadas operaciones: posible inanición (¿cuál tomar?)
- Estado alcanzable del que no sale ninguna transición: bloqueo
- Ciclos que respetan *modo de uso*: deberían dar equitatividad

Tablas y grafo *buffer* acotado

Operación	**CPRE**	**POST$^+$**
Poner(d)	$n < MAX$	$n^{sal} = n^{ent} + 1 \wedge n^{ent} \geq 0$
Tomar(d)	$n > 0$	$n^{sal} = n^{ent} - 1 \wedge n^{ent} \leq MAX$

- n: número elementos en *buffer*
- Tomar/Poner: desbloqueo mutuo asegurado
- Tomar/Tomar y Poner/Poner: desbloqueo no asegurado



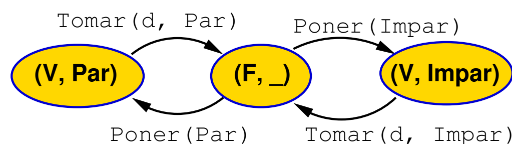
- No bloqueo: siempre alguna transición aplicable
- Vivacidad: siempre oportunidad para todas las operaciones (al llegar al final, sólo una aplicable)

Buffer par/impar

Operación	CPRE	POST+
Poner(D)	$\neg HayDato$	$HayDato^{sal} \wedge Dato^{sal} = d$
Tomar(D, P) $P \mapsto Par$	$HayDato \wedge Dato \bmod 2 = 0$	$\neg HayDato^{sal}$
Tomar(D, P) $P \mapsto Impar$	$HayDato \wedge Dato \bmod 2 = 1$	$\neg HayDato^{sal}$

- ¿Número infinito de estados?
- No realmente: precondiciones (→ transiciones) sólo dependen de paridad
- Reflejar sólo las **clases** de estados que dan diferentes valor de verdad en las precondiciones

Buffer par/impar (Cont.)

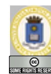


Donde:

$$\texttt{Tomar(d, Par)} \equiv \texttt{Tomar(d, p)} \wedge \texttt{p = Par}$$
$$\texttt{Poner(Par)} \equiv \texttt{Poner(p)} \wedge \texttt{p mod 2 = 0}$$
$$\texttt{(V, Par)} \equiv estado = \texttt{(v, p)} \wedge \texttt{v = cierto} \wedge \texttt{p mod 2 = 0}$$

- Nodos abstraen estado recurso
- Sólo lo necesario para decidir si bloquear o no
- Argumentos en llamadas resumen lo necesario para decidir
- Abstracción de argumentos / estado depende del problema

Multibuffer



- Generalización *buffer*:
  - Tomar/Poner especifican cuántos elementos:

    M: Multibuffer
    S: Secuencia a Tomar / Poner
    N: Número de elementos

    $\begin{cases} \text{Tomar}(M, S, N) \\ \text{Poner}(M, S, N) \end{cases}$

    (O Longitud*(S)* determina cuántos)
  - Cada una de ellas atómica: no equivale a `N` Poner/Tomar
- Contenido *Buffer* no importante
- Sólo cuántos en multibuffer y cuántos a poner/tomar
- Formalización inmediata

Modelo *multibuffer*

ACCIÓN Poner: $Tipo_MB[es] \times Tipo_MB[e] \times \mathbb{N}[e]$
ACCIÓN Tomar: $Tipo_MB[es] \times Tipo_MB[s] \times \mathbb{N}[e]$

TIPO: $Tipo_MB = \text{Secuencia}(Tipo_Dato)$

INVARIANTE: $\forall b \in Tipo_MB \bullet \text{Longitud}(b) \leq MAX$

INICIAL(b): $b = \langle\rangle$

CPRE: $\text{Longitud}(b) \geq n$
Tomar(b, s, n)
POST: $b^{ent} = b^{sal} + s^{sal}(1..n)$

CPRE: $\text{Longitud}(b) + n \leq MAX$
Poner(b, s, n)
POST: $b^{sal} = s^{ent}(1..n) + b^{ent}$

- Fácil de implementar (familia de *entries*: `multibuffer_seg.adb`)

Multibuffer: ¿estado de bloqueo?

- ¿Puede haber una implementación *resistente al interbloqueo*?
- ¡Depende de requisitos iniciales!
- Ejemplo: sólo 2 procesos, $Max = 10$, $b = \langle 5 \rangle$

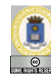
  Poner(b, s, 6) Tomar(b, s, 6)
- Revisar condiciones de uso → CPRE, tipos:

ACCIÓN Poner: $Tipo_MB[es] \times Tipo_MB[e] \times Tipo_LS[e]$

TIPO: $Tipo_MB = \text{Secuencia}(Tipo_Dato)$
$Tipo_LS = 1..\lfloor \frac{Max}{2} \rfloor$

- Cambio anterior también válido en PRE (pero no en CPRE)

Multibuffer: ¿estado de bloqueo? (Cont.)



- Debemos evitar que haya Tomar suspendido y Poner suspendido
- En general debe asegurarse que una operación tiene su guarda abierta
- Vamos a probarlo: Tomar(B,S,N) y Poner(B,S,M)
- Recordatorio: por **PRE**, $M \leq \lfloor \frac{Max}{2} \rfloor$ y $N \leq \lfloor \frac{Max}{2} \rfloor$
- Veremos que Tomar suspendido → Poner no suspendido

$$\frac{\mathbf{PRE}(\text{Tomar}) \quad \wedge \quad \neg\mathbf{CPRE}(\text{Tomar})}{}$$

$$n \leq \lfloor \tfrac{Max}{2} \rfloor \quad \wedge \quad \neg(\text{Longitud}(b) \geq n)$$
$$n \leq \lfloor \tfrac{Max}{2} \rfloor \quad \wedge \quad \text{Longitud}(b) < n$$
$$\text{Longitud}(b) < n \leq \lfloor \tfrac{Max}{2} \rfloor$$
$$m + \text{Longitud}(b) < m + n \quad \leq \quad m + \lfloor \tfrac{Max}{2} \rfloor \leq Max$$

Multibuffer: tabla bloqueos

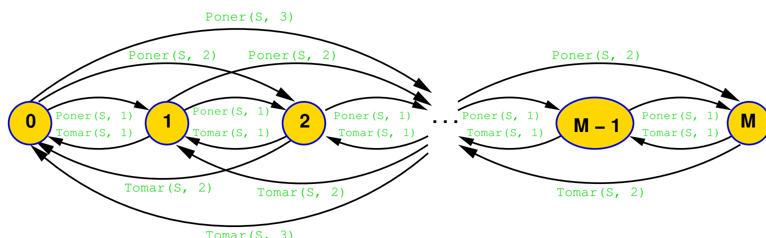
Operación	C-PRE inf.	dep.	C-PRE cod.
Poner(m, s, n)	Hay sitio	**si**	$1 + \text{Longitud}(m) \leq Max$
			$\vdots$
			$\lfloor \frac{Max}{2} \rfloor + \text{Longitud}(m) \leq Max$
Tomar(m, s, n)	Hay elementos	**sí**	$\text{Longitud}(m) \geq 1$
			$\vdots$
			$\text{Longitud}(m) \geq \lfloor \frac{Max}{2} \rfloor$

- No necesitamos estudiar los casos prohibidos por la **PRE** o por el dominio

Multibuffer: CPRE/POST$^+$

Operación	CPRE	POST$^+$
Poner(b,s,n)	$\text{Longitud}(b) + n \leq MAX$	$\text{Longitud}(b^{sal}) = \text{Longitud}(b^{ent}) + n \wedge b^{sal} = s^{ent}(1..N) + b^{ent} \wedge$ $\text{Longitud}(b^{sal}) \leq MAX \wedge$ $n \leq \frac{MAX}{2}$
Tomar(b,s,m)	$\text{Longitud}(b) \geq m$	$\text{Longitud}(b^{sal}) = \text{Longitud}(b^{ent}) - M \wedge b^{ent} = b^{sal} + s^{sal}(1..M) \wedge$ $\text{Longitud}(b^{sal}) \leq MAX \wedge$ $m \leq \frac{MAX}{2}$

Grafo estados y transiciones



- Sólo número elementos relevante
- Elección entre operaciones distintas: similar a *buffer*
- Elección entre variantes de una operación como cuenta bancaria (se favorece un extremo)
- Atención por orden llegada en cada operación

Reserva y devolución de recursos

- Clase general de problemas: necesidad de M unidades de un recurso
- Muchas veces: simetría necesitar / devolver (o simetría $>/<$)

Tamaño petición		Limitado	Ilimitado
	≥ 1	*multibuffer*	cuenta compartida
	$= 1$	*buffer*	aparcamiento

Cantidad almacenable

Gestor de memoria

- Modelización de manejo de memoria
- Solicitar y devolver memoria
- Solicitar devuelve puntero a zona
- Procesos **suspenden** hasta que haya memoria
- Supondremos que devolución necesita puntero anterior y tamaño
- Protocolo: cualquier solicitud lleva aparejada una devolución en algún momento (quizás tras la muerte del proceso)
- Modelización mediante mapa de bits (por ejemplo, por páginas de memoria)

Gestor de memoria: tipos

C-TADSOL Gestor de memoria
OPERACIONES
ACCIÓN Solicitar: *Tipo_GM[es]* $\times$ *Tipo_Tam[e]* $\times$ *Tipo_Dir[s]*
ACCIÓN Devolver: *Tipo_GM[es]* $\times$ *Tipo_Tam[e]* $\times$ *Tipo_Dir[e]*

SEMÁNTICA
DOMINIO:
TIPO: *Tipo_GM* = Secuencia($\mathbb{B}$)
Tipo_Tam = *1..Max*
Tipo_Dir = *1..Max*

INVARIANTE: $\forall m \in Tipo_GM \bullet \text{Longitud}(m) = Max$
INICIAL(m): $\forall i, 1 \leq i \leq Max \bullet \neg m(i)$

- Mapa de booleanos para posiciones libres
- Parecido tipos con *multibuffer*

Gestor de memoria: operaciones

CPRE: $\exists k, 1 \leq k \leq Max - n + 1 \bullet (\forall j, k \leq j < k + n \bullet \neg m(j))$
Solicitar(m, n, i)
POST: $m^{sal} = m^{ent} \setminus \exists k, 1 \leq k < Max - n + 1 \bullet (\forall j, k \leq j < k + n \bullet (\neg m^{ent}(j) \wedge m^{sal}(j)) \wedge i^{sal} = k)$

PRE: $\forall k, i \leq k < n + i \bullet m(k)$
CPRE: cierto
Devolver(m, n, i)
POST: $m^{ent} = m^{sal} \setminus \forall k, i \leq k < n + i \bullet \neg m^{sal}(k)$

- **Solicitar** busca n posiciones libres adyacentes y las marca ocupadas
- **Devolver** las reinstaura como libres

Gestor: bloqueos

Operación	C-PRE inf.	dep.	C-PRE cod.
Solicitar(m,n,a)	n pos. libres	**si**	$\exists k, 1 \leq k \leq Max \bullet \neg m(k)$
			$\exists k, 1 \leq k \leq Max - 1 \bullet \neg m(k) \wedge \neg m(k+1)$
			$\vdots$
			$\neg m(1) \wedge \neg m(2) \wedge \cdots \wedge \neg m(Max)$
Devolver(m,n,a)	—	**no**	*cierto*

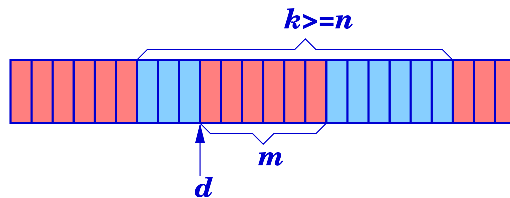
Gestor: CPRE/POST$^+$

Operación	**CPRE**	**POST$^+$**
Solicitar(m,n,i)	$\exists k, 1 \leq k \leq Max - n + 1 \bullet (\forall j, k \leq j < k + n \bullet \neg m(j))$	$m^{sal} = m^{ent} \setminus \exists k, 1 \leq k < Max - n + 1 \bullet (\forall j, k \leq j < k + n \bullet (\neg m^{ent}(j) \wedge m^{sal}(j)) \wedge i^{sal} = k)$
Devolver(m,n,i)	cierto	$m^{ent} = m^{sal} \setminus \forall k, i \leq k < n + i \bullet \neg m^{sal}(k)$

- Grafo: similar a *multibuffer*
- Estado y condiciones concurrencia más elaboradas
- Pero transiciones posibles iguales

Gestor: implementación

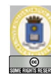
- Similar cuenta bancaria y multibuffer
- Puede usarse el mismo esquema *uno a uno*
- Precondición recorre todo el mapa de la memoria
- ¿Puede hacerse más barato?
  - Comprobar tamaño memoria liberada al devolver (pueden unirse bloques libres)
- Arrancar bloqueada n tras liberar dirección d que devuelve m sólo necesita observar *alrededores* de d



Desbloqueos explícitos

- Justificados por eficiencia
- Aún en caso de usar atención *uno a uno*
- Sólo una variable de bloqueo
- Devoluciones comprueban si bloqueada puede rearrancar
- **No realmente necesario despertar en cascada** (porque a lo sumo un proceso bloqueado)

Parte desbloqueada



```
entry Solic (Long: in T_Dir;
             Dir: out T_Dir)
when not Bloq is
begin
  Bloq:= True;
  Cpre_Solic:= Hay_Espacio(Long);
  Registro_Long:= Long;
  requeue Solic_Aplazado;
end Solic;

entry Solic_Aplazado
        (Long: in T_Dir;
         Dir :out T_Dir)
when Cpre_Solic is
begin
  Bloq:= False;
  Hallar_Espacio(Long, Dir);
end Solic_Aplazado;
```

- Sólo uno bloqueado en cada momento
- ¿Quién desbloquea?

Parte desbloqueadora

```
entry Devolver (Long: in  T_Dir;
                Dir: out T_Dir)
when True is
begin
  Devolver_Espacio(Long, Dir);
  if Bloq and
    Hay_Espacio(Registro_Long) then
    Cpre_Solic:= True;
  end if;
end Devolver;
```

- Despertar sólo si bloqueado
- Si bloqueado: sólo un Devolver puede rearrancar
- En otros casos: desbloqueo en cascada

- Devolver_Espacio puede determinar directamente si hay Registro_Long posiciones libres: Dev_Esp(Long, Dir, Libres)
- Devolver_Espacio y Hay_Espacio pueden dejar puntero a inicio región libre
- ¡Cuidado con este tipo de optimizaciones!