Programación Concurrente y Distribuida

Gracias por comprar este ejemplar. Al hacerlo estás ayudando a los refugiados de Atenas, pues los beneficios se destinan íntegramente a Project Elea, un grupo de voluntarios que trabajan con los residentes del campo de refugiados de Eleonas para mejorar los estándares de vida y el bienestar comunitario dentro de este. Ver https://projectelea.org/about-us/ para más información.

En el muy improbable caso de que el proyecto deje de recibir donaciones antes de enviar el dinero, este se destinaría a otra ONG.

Bibliografía:

- Diapositivas Programación Concurrente y Distribuida, Universidad de Murcia (autor desconocido).

Capítulo 1

Introducción

La **programación concurrente** es la que se hace usando **procesos concurrentes**, que se ejecutan en la misma CPU y el sistema operativo va alternando entre ellos, y paralelos, que se ejecutan en distintas CPUs. Esta permite mejorar el uso de la CPU y por tanto la velocidad de ejecución, así como resolver problemas inherentemente concurrentes como sistemas de control, tecnologías web, interfaces de usuario o simulaciones, pero plantea problemas de sincronización y comunicación.

Hasta la aparición de Concurrent Pascal en 1972, estuvo restringida a lenguajes de bajo nivel, pero hoy en día está en auge debido a Internet, el concepto de hilo y lenguajes de programación que facilitan su uso. Estudiamos las notaciones para especificar acciones concurrentes y las técnicas para resolver problemas inherentes a la ejecución concurrente.

Distinguimos entre procesos, entidades pesadas con cambios de contexto costosos, e hilos, entidades ligeras que comparten la información del proceso y hacen menos costosos los cambios de contexto.

En sistemas monoprocesador, el sistema operativo alterna el uso de la CPU entre los procesos dando lugar a la **multiprogramación**, mientras que los multiprocesador permiten **paralelismo** real. Estos últimos son **fuertemente acoplados** si la comunicación se hace por variables compartidas, o **débilmente acoplados** si se hace por paso de mensajes, permitiendo el **procesamiento distribuido**.

Capítulo 2

Programas concurrentes

En los programas concurrentes, el orden de ejecución de las instrucciones no es total, lo que puede dar lugar a **indeterminismo**. Algunas secuencias de instrucciones no pueden ser **entremezcladas** con otras (*interleaving*), sino que su ejecución debe ser **atómica**.

Un **recurso no compartible** es aquel que varios procesos pueden usar pero no simultáneamente. Algunos de estos son la pantalla (al menos en modo texto), la impresora, el disco duro, variables de memoria, etc. Una **sección crítica** es una sección de código usada para acceder a un recurso no compartible, y debe ejecutarse en **exclusión mutua** (sólo un proceso a la vez), ejecutándose antes un **protocolo de entrada** para asegurarse de que el recurso está libre y después un **protocolo de salida** para informar al resto de que el recurso ya está libre. Se deben cumplir:

- **Progreso de la ejecución**: Si el recurso está libre, cualquier proceso podrá acceder.
- **Limitación de la espera**: Ningún proceso esperará indefinidamente para acceder.
- **Condiciones de sincronización**: Relaciones de precedencia entre acciones de procesos.

Algunas soluciones son:

- **Inhibición de interrupciones** durante las secciones críticas en sistemas monoprocesador. Puede degradar el sistema y hace que los usuarios tengan que controlar manualmente las interrupciones.
- **Espera ocupada**: Los procesos esperan para entrar a la sección crítica comprobando continuamente el valor de algunas variables, según un algoritmo como el de **Peterson** para dos procesos, llamados 0 y 1, donde `p` indica el proceso actual.

  $E[p] \leftarrow \text{true}$
  $turn \leftarrow 1 - p$
  while $C[1-p]$ **and** $turn = 1 - p$ **do**
  end while
  $E[p] \leftarrow \text{false}$

Los programas concurrentes no tienen por qué acabar. Su corrección pasa por cumplir propiedades de seguridad, como exclusión mutua; condiciones de sincronización, y no dar lugar a interbloqueo pasivo o *deadlock*; y propiedades de vivacidad, como no dar lugar a interbloqueo activo o *livelock*, o a inanición.

Capítulo 3

Semáforos

En la espera ocupada, no hay separación entre las variables usadas para sincronización y para computación; se hace un uso ineficiente del procesador, y es difícil diseñar, comprender y probar la corrección de los programas.

La necesidad de protocolos correctos y bloqueo de procesos lleva a Dijkstra a introducir en 1965 el concepto de **semáforo** (semaphore), de los que hay dos tipos: **binarios**, que admiten los valores 0 y 1, y **generales**, que admiten cualquier entero no negativo. Operaciones:

- **Inicialización** (initial(semaphore, integer)): Inicializa un semáforo a un valor.
- **Espera de señales** (wait(semaphore)): Si el valor del semáforo es 0, se bloquea esperándolo; de lo contrario disminuye el valor en 1.
- **Señalización** (signal(semaphore)): Si hay un proceso esperando al semáforo, lo desbloquea; de lo contrario aumenta el valor en 1.

Podemos verlo como que el semáforo tiene un número inicial de permisos, de modo que con wait un hilo adquiere un permiso del semáforo y con signal lo devuelve.

- Para exclusión mutua, usamos un semáforo binario inicializado a 1, necesitando un permiso para entrar en la sección crítica. Llamamos **cerrojo** o *mutex* a un semáforo con este uso.
- Si un proceso debe esperar a otro, usamos un semáforo inicializado a 0, y el proceso que espera adquiere un permiso del semáforo que debe dar el otro.
- Si un proceso debe esperar a muchos, debe recibir tantos permisos como procesos a esperar.
- Si muchos procesos esperan a uno, este debe dar tantos permisos como procesos le esperen.

3.1. Productor/consumidor

En este problema, un proceso **productor** va produciéndo elementos e insertándolos en una cola, y un proceso **consumidor** va consumiendo y procesando los elementos en la cola, que

se supone que tiene un máximo de N elementos. La cola es un recurso no compartible y la inserción y extracción en ella son secciones críticas. Antes de insertar debe haber un hueco y antes de extraer debe haber algún elemento.

queue, of size N
empty, semaphore for free slots (N).
full, semaphore for empty slots (0).
mutex, lock for several readers/writers (1).

process type PRODUCER:
    ...
    wait(*empty*)
    wait(*mutex*)
    Insert *item* in *queue*
    signal(*mutex*)
    signal(*full*)
    ...
end process

process type CONSUMER:
    ...
    wait(*full*)
    wait(*mutex*)
    Get *item* from *queue*
    signal(*mutex*)
    signal(*empty*)
    ...
end process

3.2. Lectores/escritores

Hay una variable compartida en la que varios procesos escriben y varios leen. La variable es un recurso no compartible y los accesos de escritura son secciones críticas, pero las lecturas no pueden ejecutarse a la vez que las de lectura. No se puede leer la variable mientras otro proceso la escribe. Hay dos opciones.

3.2.1. Prioridad lectores

Un lector no espera porque un escritor esté esperando.

readers, number of current readers (0).
read, lock for reading (1).
write, lock for writing (1).

process type READER:
    ...
    wait(*read*)
    readers $\leftarrow$ *readers* + 1
    if *readers* = 1 **then**
        wait(*write*)
    end if
    Read resource
    wait(*mutex*)
    readers $\leftarrow$ *readers* − 1
    if *readers* = 0 **then**
        signal(*write*)
    end if
    signal(*mutex*)
    ...
end process

process type WRITER:
    ...
    wait(*write*)
    Write resource
    signal(*write*)
    ...
end process

3.2.2. Prioridad escritores

Cuando un escritor espera, ningún nuevo lector debe iniciar la lectura.

nr, number of readers (0).
nqw, number of queued writers (0).
nqr, number of queued readers (0).
writing, whether someone is writing (false).
vars, lock for variables (1).
read, sem. for reading (0).
write, sem. for writing (0).

process type READER:
    ...
    wait(*mutex*)
    if *writing* $\vee$ $nqw > 0$ **then**
        $nr \leftarrow nr + 1$
        signal(*vars*)
        wait(*read*)
        $nr \leftarrow nr - 1$
    end if
    $nr \leftarrow nr + 1$
    if $nqr > 0$ **then**
        signal(*read*)
    else
        signal(*vars*)
    end if
    Read resource
    wait(*vars*)
    $nr \leftarrow nr + 1$
    if $nr = 0 \wedge nqw > 0$ **then**
        signal(*write*)
    else
        signal(*vars*)
    end if
    ...
end process

process type WRITER:
    ...
    wait(*vars*)
    if $nr > 0 \vee$ *writing* **then**
        $nqw \leftarrow nqw + 1$
        signal(*vars*)
        wait(*write*)
        $nqw \leftarrow nqw + 1$
    end if
    writing $\leftarrow$ true
    signal(*vars*)
    Write resource
    wait(*vars*)
    writing $\leftarrow$ false
    if $nqw > 0$ **then**
        signal(*write*)
    else if $nqr > 0$ **then**
        signal(*read*)
    else
        signal(*mutex*)
    end if
    ...
end process

3.3. La comida de filósofos

N filósofos se sientan en una mesa circular, uno en cada posición, y entre cada dos posiciones hay un palillo. Los filósofos alternan entre pensar y comer, pero para comer deben coger los dos palillos a su lado.

Una solución «básica» en que cada uno toma el palillo a su izquierda y luego el de su derecha (o al revés) sufre interbloqueo si todos intentan empezar a comer a la vez. Para evitarlo suponemos que los filósofos piensan de pie y no se pueden sentar todos a la vez. En adelante los índices de listas se suponen módulo el tamaño de la lista.

seat, semaphore for seating ($N - 1$).
stick, array of N locks representing the sticks (1).

process type PHIL(*id*):
    repeat
        Think
        wait(*seat*)
        wait(*stick*[*id*])
        wait(*stick*[*id* + 1])
        Eat
        signal(*stick*[*id*])

```
        signal(stick[id + 1])
        signal(seat)
    until false
end process
```

Otra opción es tomar los dos palillos a la vez:

```
free, array of N booleans (true).
mutex, lock for free (1).

process type PHIL(id):
    repeat
        Think
        wait(mutex)
        while ¬(free[id] ∧ free[id + 1]) do
            signal(mutex)
            wait(mutex)
        end while
        free[id] ← false
        free[id + 1] ← false
        signal(mutex)
        Eat
        wait(mutex)
        free[id] ← true
        free[id + 1] ← true
        signal(mutex)
    until false
end process
```

Otra solución es la asimétrica, donde distinguimos filósofos en posiciones pares e impares:

```
process type EVENPHIL(id: even):
    repeat
        Think
        wait(stick[id + 1])
        wait(stick[id])
        Eat
        signal(stick[id + 1])
        signal(stick[id])
    until false
end process

process type ODDPHIL(id: odd):
    repeat
        Think
        wait(stick[id])
        wait(stick[id + 1])
        Eat
        signal(stick[id])
        signal(stick[id + 1])
    until false
end process
```


3.4. Implementación

```
struct semaphore
    value, an integer.
    q, a (priority) queue of threads.
end struct
function INITIAL(sem,val)
    sem.value ← val
    sem.q ← empty
end function
function WAIT(sem)
    if sem.value > 0 then
        sem.value ← sem.value − 1
    else
        Insert this thread in sem.q
        Lock this thread
    end if
end function
function SIGNAL(sem)
    if sem.q not empty then
        Take T out of sem.q
        Unlock T
    else
        sem.value ← sem.value + 1
    end if
end function
```

Las operaciones de bloqueo y desbloqueo de hilos las proporciona el sistema operativo.

Capítulo 4

Monitores

Los semáforos son complejos y de bajo nivel, y suponen que el código de sincronizacion esté distribuido entre todos los procesos. Un **monitor** está formado por:

- Un conjunto de variables privadas o permanentes, que mantienen sus valores entre llamadas y sólo están disponibles a los procedimientos del monitor.
- Un conjunto de procedimientos públicos o privados. Sólo un hilo puede ejecutar procedimientos del monitor a la vez.
- Un **cuerpo de inicialización** para inicializar ls variables permanentes.
- Colas de espera, de gestión automática.

Así se consigue encapsular los datos junto a sus operaciones de acceso, permitiendo un control estructurado de la exclusión mutua y condiciones de sincronización. Distinguimos procesos **pasivos**, que implementan los monitores y esperan a que los activos usen sus operaciones, y **activos**, que interactúan entre sí a través de las operaciones de los pasivos.

Los monitores pueden tener **variables condición** (**condition**), colas de hilos inicialmente vacías, con las operaciones:

1. Bloquear (delay(*con*)): Si hay un hilo esperando en la cola, le cede la exclusión mutua, y si no la libera. Entonces inserta el hilo actual en la cola y se bloquea. No se debe usar de forma incondicional.
2. Reanudar (resume(*con*)): Si algún hilo en la cola, desbloquea al primero de estos. Varias opciones:
   a) **Desbloquear y continuar** (**DC**): El hilo desbloqueado continúa su ejecución cuando el actual salga del monitor (puede que no inmediatamente). Normalmente implica introducir el bloqueo dentro de un bucle con lo que los procesos se pueden bloquear una vez despertados. Mesa y Java.
   b) **Retorno forzado** (**DS**): Se sale del monitor cediendo la exclusión mutua al hilo desbloqueado. Uso complejo. ConcurrentPascal.

c) **Desbloquear y esperar** (**DE**): Se bloquea el hilo actual en la cola de entrada al monitor, cediendo la exclusión mutua al hilo desbloqueado. Más fácil de usar, pero ineficiente si se reanuda al final de los procedimientos. Modula-2 y ConcurrentEuclid.

d) **Desbloquear y espera urgente** (**DU**): Como el anterior, pero el hilo actual se bloquea en una **cola de cortesía**, que tiene prioridad sobre la de entrada. Es la que supondremos en los ejemplos en pseudocódigo de este capítulo. Pascal FC y Pascal Plus.

3. ¿Vacía? (empty(*con*)): Propuesta por Hoare, indica si la cola de la condición está o no vacía.

Si el número de operaciones a ofrecer es muy alto, se suele usar el monitor para implementar un protocolo de entrada y uno de salida, como sigue:

```
monitor BinarySemaphore
    export wait, signal
    var sem: boolean (false).
        csem: condition.

    function wait
        if sem then
            delay(csem)
        else
            sem ← true
        end if
    end function
    function signal
        if ¬empty(csem) then
            resume(csem)
        else
            sem ← false
        end if
    end function
end monitor
```


4.1. Productor/consumidor

```
monitor buffer
    export insert, extract
    var q, a queue with capacity N.
        nonfull, nonempty, conditions.

    function insert(e)
        if size(q) = N then
            delay(nonfull)
        end if
        Insert e into q
        resume(nonempty)
    end function
    function extract
        if size(q) = 0 then
            delay(nonempty)
        end ifGet e from q
        resume(nonfull)
        return e
    end function
end monitor
```

4.2. Lectores/escritores

El recurso no puede estar dentro de monitor, pues debe poder ser leído por varios procesos al mismo tiempo.

4.2.1. Prioridad lectores

```
monitor RW
    export openRead, openWrite, closeRead, closeWrite
    var nr, number of readers (0).
        writing, whether someone is writing (false).
        read, write, conditions.

    function openRead
        if writing then
            delay(read)
        end if
        nr ← nr + 1
        resume(read)
    end function
    function closeRead
        nr ← nr − 1
        if nr = 0 then
            resume(write)
        end if
    end function
    function openWrite
        if nr ≠ 0 ∨ writing then
            delay(write)
        end if
        writing ← true
    end function
    function closeWrite
        writing ← false
        if ¬empty(read) then
            resume(read)
        else
            resume(write)
        end if
    end function
end monitor
```

4.2.2. Prioridad escritores

Cambian las siguientes funciones como sigue:

```
function openRead
    if writing ∨ ¬empty(write) then
        delay(read)
    end if
    nr ← nr + 1
    resume(read)
end function
```

```
function closeWrite
    writing ← false
    if ¬empty(write) then
        resume(write)
    else
        resume(read)
    end if
end function
```


4.3. La comida de filósofos

Los índices de listas son módulo el tamaño de la lista.

```
monitor table
    export takeSticks, releaseSticks
    var state, array with the state of the N philosophers, either "thinking", "eating" or "hungry" (thinking).
        sleep, array of N conditions.
        i, integer (0).

    function takeSticks(i)
        state[i] ← hungry
        test(i)
        if estate[i] ≠ eating then
            delay(sleep[i])
        end if
    end function

    function releaseSticks(i)
        state[i] ← thinking
        test(i + 4)
        test(i + 1)
```

```
    end function

    function TEST(i)
        if state[i + N − 1] ≠ eating
                ∧state[i + 1] ≠ eating
                ∧state[i] = hungry then
            state[i] ← eating
            resume(sleep[k])
        end if
    end function
```

```
end monitor

process type PHILOSOPHER(id):
    repeat
        Think
        table.TAKESTICKS(id)
        Eat
        table.RELEASESTICKS(id)
    until false
end process
```

4.4. Llamadas anidadas

Cuando un procedimiento en un monitor A hace una llamada a uno de otro B, debemos considerar si se debería retener la exclusión mutua sobre A mientras se ejecuta B, y qué ocurre cuando se ejecuta un bloqueo en B. Soluciones:

- Mantener la exclusión mutua sobre A y, al bloquearse el hilo, liberar B. Se reduce el nivel de concurrencia y puede producir interbloqueos.
- Mantener la exclusión mutua sobre ambos monitores y liberar la de los dos al bloquearse. Soluciona el interbloqueo, pero es difícil mantener el estado de A entre activaciones.
- Liberar A al hacer la llamada a B. Se gana concurrencia pero es difícil de gestionar.
- Prohibir las llamadas anidadas. Se pierde modularidad.

Capítulo 5

Paso de mensajes

Los semáforos y monitores se basan en memoria compartida, lo que no es eficiente en un sistema distribuido, donde de por sí no existe una memoria común y no conocemos el estado global, con lo que nos comunicamos por redes donde puede haber pérdida y desordenación de mensajes.

El paso de mensajes, que también se puede usar sobre memoria compartida, funciona con dos operaciones, send(*destino*, *mensaje*) y receive(*origen*, *mensaje*), con distinta semántica.

El **direccionamiento** es **directo** si emisor y receptor indican explícitamente los procesos a los que se dirigen los mensajes o de quien esperan recibir, o **indirecto** si se usa una estructura de datos intermedia.

La comunicación directa aporta seguridad y evita retardos, pero es difícil de mantener ante cambios en la denominación de los procesos, sólo permite un canal por par de procesos, hay que garantizar unicidad en la identificación y no permite aplicaciones cliente-servidor. En la indirecta distinguimos:

- **Canales**: Comunicación uno a uno. El flujo de datos puede ser unidireccional, típico en la comunicación asíncrona, o bidireccional. La capacidad puede ser cero, para comunicación síncrona; finita, en la que el llenado implica emisión bloqueante, o infinita, con el peligro de un colapso del sistema. Los mensajes pueden tener longitud fija o variable, y ser o no de un cierto tipo.

- **Puertos**: Varios a uno. Direccionamiento **asimétrico** cliente-servidor, donde el emisor debe conocer el identificador del receptor y este, al recibir el mensaje, recibe también el identificador del emisor (**direccionamiento implícito**).

- **Buzones**: Uno o varios a varios. La asociación de buzones a procesos puede ser **estática**, por declaración anticipada, o **dinámica**, usando llamadas al sistema para la conexión y la desconexión.

El paso de mensajes puede ser:

- Por valor.

- Por referencia, compartiendo una memoria común. Esto es poco seguro si hace mal y requiere control de la exclusión mutua, pero es más eficiente.

- Copia en escritura: Como por referencia, pero el sistema operativo usa el sistema de paginación para copiar la página compartida si se modifica. Usado en Mach.

Aunque consideramos que la comunnicación no sufre errores, dependiendo del caso puede haber problemas por pérdida de mensajes o ruido. El envío y recepción de mensajes puede ser bloqueante o no. Las bloqueantes son más fáciles de implementar y el envío bloqueante puede dar certeza de la recepción del mensaje. Los esquemas más comunes son:

- **Síncrono** con canales, direccionamiento indirecto, capacidad nula y flujo bidireccional.
- **Asíncrono** con buzones, direccionamiento indirecto, capacidad finita y flujo unidireccional.

Para recibir de varios buzones y evitar acoplamiento, en pseudocódigo se usa la sintaxis «select»: **select** [**when** *cond*] [**receive**|**send**](*buzón1*, *mensaje1*); *stmt** (**or** [**when** *cond*] [**receive**|**send**](*buzón*, *mensaje*); *stmt**)* [**or timeout** *N* *stmt**|**else** *stmt**] **end select**;

Se comprueban las **guardas** (los **when**) y, de entre las alternativas de envío o recepción cuya guarda se cumpla o que no tengan guarda, se elige una en que la instrucción primera no sea bloqueante. Si todas las alternativas tienen guarda y ninguna se cumple, se produce un error. Si se puede ejecutar alguna pero en todas la primera instrucción es bloqueante, se ejecuta la rama **else** si existe; de lo contrario, si existe una rama **or timeout**, se bloquea hasta que una de estas pase a no ser bloqueante, en cuyo caso se ejecuta su rama, o hasta que pasen N segundos, en cuyo caso se ejecuta la rama **or timeout**, y si no hay rama **or timeout** ni **else**, se espera indefinidamente hasta que una pase a ser no bloqueante y entonces se ejecuta esta alternativa.

Capítulo 6

Paso de mensajes asíncrono

Hay un buffer finito, un buzón (mailbox [[1..*N*]] of *tipo*), el envío no es bloqueante salvo que el buffer este lleno y la recepción tampoco salvo que esté vacío. Esto permite menos cambios de contexto y es ideal para modelos productor/consumidor, pero añade dificultad para confirmar las lecturas, en la implementación y en el tamaño apropiado del buffer.

6.1. Semáforo binario

Lo inicializamos añadiendo un mensaje a un cierto buzón, recibimos un mensaje para adquirir el semáforo y enviamos un mensaje para liberarlo.

6.2. Productor/consumidor

process type PRODUCER:
    ...
    send(*mbox*, *elem*)
    ...
end process

process type CONSUMER:
    ...
    receive(*mbox*, *elem*)
    ...
end process

6.3. Lectores/escritores

6.3.1. Prioridad lectores

Podemos adaptar la versión con semáforos o hacer lo siguiente:

process type CONTROLLER:
    var *nr*, number of readers (0).
        writing, whether someone's writing (false).

    repeat
        select when $\neg$*writing* **then**
            receive(*openRead*, *snd*)
            $nr \leftarrow nr + 1$
            send(*snd*)
        or receive(*closeRead*)
            $nr \leftarrow nr - 1$

```
        or when nr = 0 ∧ ¬writing then
            receive(openWrite, snd)
            writing ← true
            send(snd)
        or receive(closeWrite)
            writing ← false
        end select
    until false
end process
process type Reader:
    var rcv, mailbox for receiving.

    ...
    send(openRead, rcv)
    receive(rcv)
    Read
    send(closeRead)
    ...
end process
process type Writer(id):
    var rcv, mailbox for receiving.

    ...
    send(openWrite, rcv)
    receive(rcv)
    Write
    send(closeWrite)
    ...
end process
```

6.3.2. Prioridad escritores

Basta tomar el código anterior y, para la rama que lee de *openRead*, añadir la condición de que *openWrite* no debe tener solicitudes (mensajes) pendientes. Otra forma es:

6.4. La comida de filósofos[1]

N es el número de filósofos, y los índices de las listas son módulo el tamaño de la lista.

```
process type Philosopher(id):
    var rcv, mailbox for receiving.

    repeat
        Think
        send(acquire[id], rcv)
        receive(rcv)
        Eat
        send(release, id)
    until false
end process
process type Controller:
    var sticks, whether each stick is free (all
true).

    repeat
        select when sticks[0] ∧ sticks[1]
then
            receive(acquire[0], snd)
            sticks[0] ← false
            sticks[1] ← false
            send(snd)
        or when sticks[1] ∧ sticks[2] then
            ...
        or when sticks[N] ∧ sticks[1] then
            receive(acquire[N], snd)
            sticks[N] ← false
            sticks[1] ← false
            send(snd)
        or receive(release, id)
            sticks[id] ← true
            sticks[id + 1] ← true
        end select
    until false
end process
```

[1] En esta sección, y en todo el capítulo siguiente, cada vez que se usa k dentro de un **select**, existen N cláusulas **select**, una por proceso del tipo correspondiente, en las que $k = 1, \ldots, N$.

Capítulo 7

Paso de mensajes síncrono

Se usa un canal bidireccional (channel of *tipo*), y el envío y la recepción son bloqueantes con sincronización entre emisor y receptor llamada **rendez-vous**. Para hacer un envío bloqueante a partir de uno no bloqueante, el emisor envía al receptor un buzón y espera a este, mientras que el receptor, al recibir el mensaje, envía una confirmación por dicho buzón.

7.1. Semáforo general

process type SEMAPHORE(init):
  var *val*, the value of the semaphore (*init*).

  repeat
    select when $val > 0$ **then**
      receive($wait[k]$)
      $val \leftarrow val - 1$
    or
      receive($signal[k]$)
      $val \leftarrow val + 1$
    end select
  until false
end process

7.2. Productor/consumidor

process type CONTROLLER:
  var *buffer*, with capacity N.

  repeat
    select when *buffer* not full **then**
      receive($insert[k]$, *elem*)
      Insert *elem* into *buffer*
    or when *buffer* not empty **then**
      send($extract[k]$, head of the queue)
      Take the head out of the queue
    end select
  until false
end process
process type PRODUCER(id):
  ...
  send($insert[id]$, *elem*)
  ...
end process
process type CONSUMER(id):
  ...
  receive($extract[id]$, *elem*)
  ...
end process

7.3. Lectores/escritores

7.3.1. Prioridad lectores

process type CONTROLLER:
  var nr, number of readers (0).
    writing, whether someone's writing (false).

  repeat
    select when $\neg$*writing* **then**
      receive(*openRead*[k])
      $nr \leftarrow nr + 1$
    or
      receive(*closeRead*[k])
      $nr \leftarrow nr - 1$
    or when $nr = 0 \wedge \neg$*writing* **then**
      receive(*openWrite*[k])
      writing $\leftarrow$ true
    or ▷ For $k \in \{1, \ldots, N\}$
      receive(*closeWrite*[k])
      writing $\leftarrow$ false
    end select
  until false
end process

7.3.2. Prioridad escritores

process type CONTROLLER:
  var nr, number of readers (0).
    nqw, number of queued writers (0).
    writing, whether someone's writing (false).

  repeat
    select when $nqw = 0 \wedge \neg$*writing* **then**
      receive(*openRead*[k])
      $nr \leftarrow nr + 1$
    or
      receive(*closeRead*[k])
      $nr \leftarrow nr - 1$
    or
      receive(*requestWrite*[k])
      $nqw \leftarrow nqw + 1$
    or when $nr = 0 \wedge \neg$*writing* **then**
      receive(*openWrite*[k])
      writing $\leftarrow$ true
    or
      receive(*closeWrite*[k])
      writing $\leftarrow$ false
    end select
  until false
end process
process type READER(id):
  ...
  send(*openRead*[id])
  Read
  send(*closeRead*[id])
  ...
end process
process type WRITER(id):
  ...
  send(*requestWrite*[id])
  send(*openWrite*[id])
  Write
  send(*closeWrite*[id])
  ...
end process

7.4. La comida de filósofos

Los índices para acceso a listas son módulo N.

process type PHILOSOPHER(id):
    repeat
        Think
        send($acquire[id]$)
        Eat
        send($release[id]$)
    until false
end process
process type CONTROLLER:
    var *sticks*, whether each stick is free (all true).

    repeat
        select when $sticks[k] \wedge sticks[k+1]$ **then**
            receive($acquire[k]$)
            $sticks[k] \leftarrow$ false
            $sticks[k+1] \leftarrow$ false
        or
            receive($release[k]$)
            $sticks[k] \leftarrow$ true
            $sticks[k+1] \leftarrow$ true
        end select
    until false
end process

Capítulo 8

Invocación remota

La invocación remota se basa en el paso de mensajes síncrono, con capacidad nula y flujo bidireccional, para simular llamadas a procedimientos de otro proceso. Se hace un *extended rendez-vous*, en que el receptor queda bloqueado hasta que le llama un cliente y este se bloquea hasta que el receptor recibe el mensaje y termina el procedimiento. Es un esquema asímetrico, en el que el emisor debe conocer la identidad del receptor pero no al revés. Los servidores publican sus puntos de entrada, y si varios clientes quieren ejecutar el mismo servicio, sólo se atiende a uno.

En pseudocódigo, declaramos los puntos de entrada de un proceso con **entry** *nombre*(*parámetro**). La sintaxis **accept** *nombre*(*parámetro**) **do** *stmt* sirve para recibir y ejecutar una llamada al procedimiento con un cierto *nombre* y ejecutar *stmt* cuando se llame, y «select» se modifica para aceptar una cláusula de este tipo donde iría el **send** o el **receive**.

8.1. Semáforos

```
process type SEMAPHORE(init):
    var val, the semaphore value (init).

    repeat
        select when val > 0 then
            accept WAIT do
                val ← val − 1
            end accept
        or
            accept SIGNAL do
                val ← val + 1
            end accept
        end select
    until false
end process
```

8.2. Productor/consumidor

```
process type BUFFER:
    var q, an initially empty queue.

    repeat
        select when buf not empty then
            accept EXTRACT do
                return Extract from q.
            end accept
        or when buf not full then
            accept INSERT(elem) do
```

Insert *elem* to q.
end accept
end select
until false
end process

8.3. Lectores/escritores

8.3.1. Prioridad lectores

process type CONTROLLER:
var nr, number of readers (0).
writing, whether someone's writing (false).

repeat
select when $\neg writing$ **then**
accept OPENREAD **do**
$nr \leftarrow nr + 1$
end accept
or
accept CLOSEREAD **do**
$nr \leftarrow nr - 1$
end accept
or when $nr = 0 \wedge \neg writing$ **then**
accept OPENWRITE **do**
$writing \leftarrow$ true
end accept
or
accept CLOSEWRITE **do**
$writing \leftarrow$ false
end accept
end select
until false
end process

8.3.2. Prioridad escritores

process type CONTROLLER:
var nr, number of readers (0).
nqw, number of queued writers (0).
writing, whether someone's writing (false).

repeat
select when $\neg writing \wedge nqw = 0$ **then**
accept OPENREAD **do**
$nr \leftarrow nr - 1$
end accept
or
accept CLOSEREAD **do**
$nr \leftarrow nr - 1$
end accept
or
accept REQUESTWRITE **do**
$nqw \leftarrow nqw + 1$
end accept
or when $nr = 0 \wedge \neg writing$ **then**
accept OPENWRITE **do**
$writing \leftarrow$ true
$nqw \leftarrow nqw - 1$
end accept
or
accept CLOSEWRITE **do**
$writing \leftarrow$ false
end accept
end select
until false
end process

8.4. Comida de filósofos

Hay un total de N filósofos, y los índices de acceso a listas son módulo N.

```
process type STICK:
    repeat
        accept TAKE do
        end accept
        accept RELEASE do
        end accept
    until false
end process
global sticks, array of N sticks.

process type TABLE:
    var eating, number of people eating (0).

    repeat
        select when eating < N − 1 then
            accept SIT do
                eating ← eating − 1
            end accept
        or
            accept GETUP do
                eating ← eating − 1
            end accept
        end select
    until false
end process
process type PHILOSOPHER(id):
    repeat
        Think
        TABLE.SIT
        sticks[id].TAKE
        sticks[id+1].TAKE
        Eat
        sticks[id].RELEASE
        sticks[id + 1].RELEASE
        TABLE.GETUP
    until false
end process
```