Sistemas Concurrentes y Distribuidos.

Guía, apuntes, problemas, seminarios y prácticas.

Curso 2015-16

Contents

Part I

Guía de la asignatura

Chapter 1

Guía de la asignatura

1.1 Datos generales

Datos generales de la asignatura.

nombre	**Sistemas Concurrentes y Distribuidos**
titulación	Grado en Informática
	Grado en Informática y Matemáticas
tipo	obligatoria (materia común de la rama) (GI)
curso y cuatrimestre	2º curso, 1er cuatrim. (GI)
adscrita al departamento	Lenguajes y Sistemas Informáticos
créditos totales	6 ECTS
horas semanales presen.	4 horas (2 teoría + 2 prácticas)
horas sem. no presen.	4 horas
más información	http://lsi.ugr.es/scd
	https://tutor2.ugr.es
	http://lsi.ugr.es/lsi/node/943

Recursos en la Web

- Página web de la asignatura con documentación para alumnos:
  - http://lsi.ugr.es/scd (login: **aluscd**, password: **pwdscd1314**)
- Sistema web para gestión de grupos y calificaciones (sistema Tutor)
  - pág. principal: https://tutor.ugr.es
  - ficha asignatura: https://tutor.ugr.es/functions/index.php/idop_950/subject_100
- Sitio web del departamento:
  - pág. principal: http://lsi.ugr.es
  - ficha de la asignatura: http://lsi.ugr.es/lsi/node/943
  - profesorado depto.: http://lsi.ugr.es/lsi/personal

Grupos grandes (teoría)

Gr	Día	Horas	Profesor
A	-	-	Carlos Ureña Almagro
B	-	-	Manuel Noguera García
C	-	-	Carlos Ureña Almagro
D	-	-	Pedro Villar Castro

Grupos pequeños (prácticas).

Gr	Día	Hora	Profesor
A1			C.Ureña
A2			C.Ureña
A3			A.Sánchez
B1			P.Villar
B2			M.Noguera
B3			A.Sánchez
C1			P.Villar
C2			G.Guerrero
C3			R.Prieto
D1			P.Villar
D2			G.Guerrero

1.2 Objetivos y temario

Objetivos.

- Comprender la importancia de la programación concurrente en las aplicaciones de hoy en día.
- Identificar las principales características de los distintos tipos de sistemas concurrentes que existen.
- Conocer y entender los problemas que plantea el desarrollo de programas concurrentes, y que no aparecen en la programación secuencial.
- Entender los conceptos de sincronización y exclusión mutua entre procesos.
- Identificar las propiedades de seguridad y vivacidad que un sistema concurrente debe cumplir y ser capaz de razonar si dichas propiedades se cumplen.
- Conocer los principales modelos de programación concurrente, paralela y distribuida.
- Adquirir experiencia y conocimiento en los mecanismos de sincronización y comunicación que se utilizan en la actualidad para desarrollar programas concurrentes tanto para sistemas de memoria compartida como para sistemas distribuidos.
- Entender el funcionamiento de semáforos y monitores como mecanismos de sincronización para memoria compartida y comprender cómo se pueden resolver problemas de programación concurrente usando monitores.

- Ser capaz de desarrollar algoritmos para sistemas basados en memoria compartida y para sistemas distribuidos que resuelvan problemas modelo en programación concurrente.
- Conocer y ser capaz de usar bibliotecas y plataformas estandarizadas para la implementación de programas concurrentes basados en memoria compartida y para sistemas distribuidos
- Conocer las técnicas más destacadas para el diseño de sistemas de tiempo real

Temario de teoría

Se divide en los siguientes capítulos:

1. Introducción a la Programación Concurrente.
2. Algoritmos y mecanismos de sincronización basados en memoria compartida.
3. Sistemas basados en paso de mensajes.
4. Introducción a los sistemas de tiempo real.

Temario de prácticas

Prácticas evaluables:

1. Resolución de problemas de sincronización con semáforos.
2. Programación de monitores con hebras.
3. Programación de aplicaciones distribuidas.
4. Programación de tareas periódicas con prioridades.

Seminarios impartidos presencialmente en las clases de prácticas:

1. Introducción a la programación mutihebra usando semáforos.
2. Introducción a la programación mutihebra con monitores.
3. Introducción al uso de una interfaz de paso de mensajes.

1.3 Metodología docente y evaluación

Metodología docente.

Dado el carácter básico de la asignatura, se pretende que el alumno adquiera los conocimientos teóricos de la materia y los sepa aplicar con soltura. Para ello, las actividades de enseñanza-aprendizaje que se realizarán serán una combinación de:

- Actividades presenciales, entre las que se incluyen:
  - Clases magistrales

  - Resolución de ejercicios/problemas individuales y/o en grupo
  - Sesiones prácticas en laboratorio
  - Tutorías
  - Pruebas objetivas
- Actividades no presenciales, que pueden ser:
  - Estudio individual o en grupo
  - Realización de ejercicios/problemas/trabajos tanto individuales como en grupo
  - Confección de la carpeta de aprendizaje o portafolio de prácticas.

Calificación

Los criterios básicos para obtener la calificación son los siguientes:

- El peso de la teoría será del 65% y el de las prácticas del 35%
- Para aprobar la asignatura es necesario tener una calificación numérica **superior o igual a 5** (sobre 10), sumando teoría y prácticas.
- Además del requisito anterior, se establece como requisito adicional para superar la asignatura que, tanto la calificación correspondiente a la parte teórica como la correspondiente a la parte práctica, sean **cada una de ellas sea igual o superior al 40%** de la nota máxima de dicha parte.

Evaluación de teoría y/o prácticas ya superadas

Existen estas opciones:

Para los alumnos que no logren aprobar la asignatura en la convocatoria ordinaria de febrero (o septiembre) del año 2016

Si el alumno obtiene una nota **igual o superior al 40% en teoría o prácticas**, se podrá guardar dicha nota para la convocatoria de septiembre (o diciembre) del año 2016.

Para los alumnos que no logren aprobar la asignatura en el curso académico 2015-16

Si el alumno obtiene una nota **igual o superior al 50% en prácticas con evaluación continua**, se podrá guardar dicha nota para todas las convocatorias de cursos posteriores.

Modalidades de evaluación

Los alumnos pueden seleccionar una de las siguientes dos modalidades de evaluación:

- **Evaluación continuada y formativa:** Basada en la asistencia regular a clases de teoría y prácticas y la realización de las actividades propuestas.
- **Examen final:** Basada en la realización de un examen escrito relativo al temario de teoría, y la defensa de las prácticas ante el profesor de prácticas asignado al alumno.

Evaluación continuada y formativa

En esta modalidad, los alumnos tienen que asistir a clase regularmente y realizar las pruebas y ejercicios que se plantean a lo largo del curso.

- Deben asistir a todas las pruebas objetivas de teoría o prácticas.
- Se admite que, de forma justificada, se falte a una prueba como máximo.
- Esta modalidad será el mecanismo de evaluación por defecto de todos los alumnos salvo para los que soliciten de forma justificada (y les sea concedida) la otra modalidad (evaluación única final)

Calificación de teoría en evaluación continua

La calificación de **teoría** (6,5 puntos) se reparte de la siguiente forma:

- 6.5 puntos corresponden a 3 pruebas objetivas individuales realizadas al final de cada tema. La distribución de la puntuación por temas es la dada en la siguiente tabla:

Tema	1	2	3 y 4	Total
Punt. máx.	1.2	2.3	3	6.5

- Se pueden 0.5 puntos adicionales correspondiente a la resolución de ejercicios, problemas, y/o trabajos (se podrá tener en cuenta la asistencia a clases de teoría), hasta un máximo de 6.5 en teoría.

Calificación de prácticas en evaluación continua

La calificación de **prácticas** (3,5 puntos) se obtiene de la siguiente forma:

- 3.5 puntos correspondientes a cinco pruebas objetivas (prácticas 1, 2, 3, 4 y 5) que se realizan durante la última sesión de prácticas de la parte a evaluar. Esta a su vez se distribuye como indica la siguiente tabla:

Tema	1	2	3	4	Total
Punt. máx.	0.9	1.2	1.1	0.3	3.5

- Se pueden obtener 0.5 puntos por las soluciones de los ejercicios propuestos que sean presentadas en clase a los compañeros, previo acuerdo con el profesor, con un máximo de 3.5 puntos en prácticas.

La carpeta de aprendizaje podrá ser revisada al menos una vez durante el cuatrimestre en tutorías

Evaluación por examen final

Para aquellos alumnos que hayan solicitado y les haya sido concedida la evaluación por esta modalidad. Consta de dos partes:

- **Examen de teoría**: tendrá lugar en la fecha fijada por el centro
- **Entrega, defensa de las prácticas, y resolución de casos prácticos**: tendrá lugar en las fechas en las que se reserven para este fin las aulas de prácticas de la ETSIIT (puede ser el mismo día del examen)

La evaluación de teoría y prácticas en las convocatorias de septiembre y diciembre de 2015 se realizará de igual forma que la descrita aquí para evaluación por examen final.

1.4 Bibliografía

Bibliografía fundamental (1/2)

- **G. R. Andrews.** *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 2000.
- **M. Ben-Ari.** *Principles of Concurrent and Distributed Programming.* Prentice Hall, 2nd edition. 2006.
- **J. T. Palma, C. Garrido, F. Sánchez, A. Quesada.** *Programación Concurrente.* Thomson-Paraninfo. 2003.
- **G. R. Andrews.** *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.
- **F. Almeida, D. Gimenez, J. M. Mantas, A.M. Vidal.** *Introducción a la Programacion Paralela.* Paraninfo Cengage Learning, 2008.

Bibliografía fundamental (2/2)

- **V. Kumar , A. Grama, A. Gupta, G. Karypis.** *Introduction to Parallel Computing.* Benjamin/Cummings Publishing Company, 2003.
- **N. Santoro.** *Design and analysis of distributed algorithms.* Wiley Series on parallel and distributed computing. 2007.
- **A. Burns, A. Wellings.** *Sistemas de Tiempo Real y Lenguajes de Programación.* (3ª edición). Addison Wesley, 2003.

Bibliografía complementaria.

- **N. Gehani, A.D. McGettrick.** *Concurrent Programming.* International Computer Science Series. Addison-Wesley. 1988.
- **C. Hughes, T. Hughes.** *Professional Multicore Programming: Design and Implementation for C++ Developers.* Wrox Programmer to Programmer. 2008.
- **C. Breshears.** *The Art of Concurrency: A Thread Monkeyś Guide to Writing Parallel Applications.* OŔeilly Media. 2009.
- **N.A. Lynch.** *Distributed Algorithms.* Morgan Kaufmann. 1996.

Part II

Apuntes de teoría

Chapter 2

Tema 1. Introducción.

2.1 Conceptos básicos y motivación

2.1.1 Conceptos básicos relacionados con la concurrencia

Programa concurrente, concurrencia y programación concurrente

- **Programa secuencial:** Declaraciones de datos + Conjunto de instrucciones sobre dichos datos que se deben ejecutar en secuencia.
- **Programa concurrente:** Conjunto de programas secuenciales ordinarios que se pueden ejecutar *lógicamente* en paralelo.
- **Proceso:** Ejecución de un programa secuencial.
- **Concurrencia:** Describe el potencial para ejecución paralela, es decir, el solapamiento real o virtual de varias actividades en el tiempo.
- **Programación Concurrente (PC):** Conjunto de notaciones y técnicas de programación usadas para expresar paralelismo potencial y resolver problemas de sincronización y comunicación.
- La PC es independiente de la implementación del paralelismo. Es una abstracción

Programación paralela, programación distribuida y programación de tiempo real

- **Programación paralela:** Su principal objetivo es acelerar la resolución de problemas concretos mediante el aprovechamiento de la capacidad de procesamiento en paralelo del hardware disponible.
- **Programación distribuida:** Su principal objetivo es hacer que varios componentes software localizados en diferentes ordenadores trabajen juntos.
- **Programación de tiempo real:** Se centra en la programación de sistemas que están funcionando continuamente, recibiendo entradas y enviando salidas a/desde componentes hardware (*sistemas reactivos*), en los que se trabaja con restricciones muy estrictas en cuanto a la respuesta temporal (*sistemas de tiempo real*).

2.1.2 Motivación de la Programación concurrente

Beneficios de la Programación concurrente

La PC resulta más complicada que la programación secuencial.

¿ Por qué es necesario conocer la Programación Concurrente ?

1. Mejora de la eficiencia

La PC permite aprovechar mejor los recursos hardware existentes.

- **En sistemas con un solo procesador:**
  - Al tener varias tareas, cuando la tarea que tiene el control del procesador necesita realizar una E/S cede el control a otra, evitando la espera ociosa del procesador.
  - También permite que varios usuarios usen el sistema de forma interactiva (actuales sistemas operativos multisusuario).
- **En sistemas con varios procesadores:**
  - Es posible repartir las tareas entre los procesadores, reduciendo el tiempo de ejecución.
  - Fundamental para acelerar complejos cómputos numéricos.

Beneficios de la Programación concurrente (2)

2. Mejora de la calidad

Muchos programas se entienden mejor en términos de varios procesos secuenciales ejecutándose concurrentemente que como un único programa secuencial.

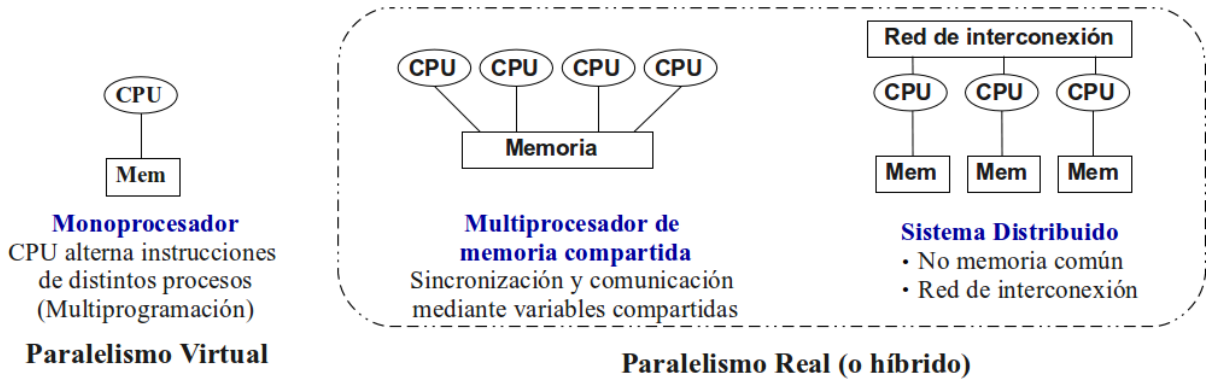
Ejemplos:

- **Servidor web para reserva de vuelos:** Es más natural, considerar cada petición de usuario como un proceso e implementar políticas para evitar situaciones conflictivas (permitir superar el límite de reservas en un vuelo).
- **Simulador del comportamiento de una gasolinera:** Es más sencillo considerar los surtidores, clientes, vehículos y empleados como procesos que cambian de estado al participar en diversas actividades comunes, que considerarlos como entidades dentro de un único programa secuencial.

2.2 Modelo abstracto y consideraciones sobre el hardware

2.2.1 Consideraciones sobre el hardware

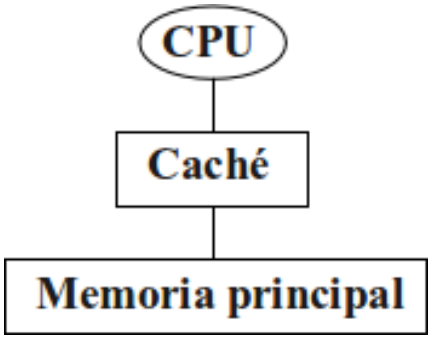
Modelos de arquitecturas para programación concurrente



Mecanismos de implementación de la concurrencia

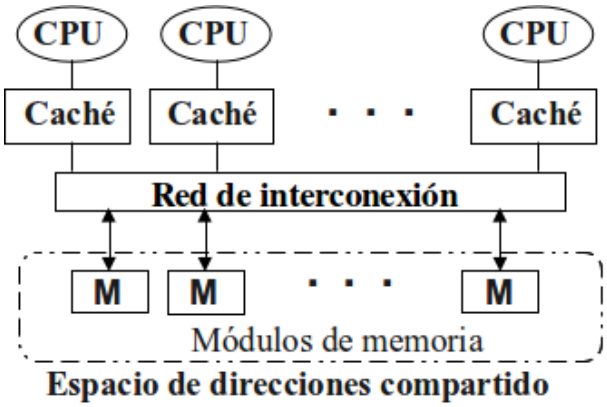
- Dependen fuertemente de la arquitectura.
- Consideran una *máquina virtual* que representa un sistema (multiprocesador o sistema distribuido), proporcionando base homogénea para ejecución de los procesos concurrentes.
- El tipo de paralelismo afecta a la eficiencia pero no a la corrección.

Concurrencia en sistemas monoprocesador



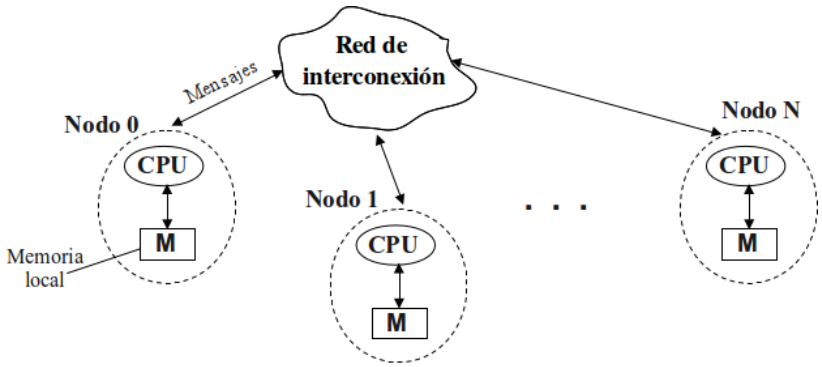
- **Multiprogramación:** El sistema operativo gestiona cómo múltiples procesos se reparten los ciclos de CPU.
- Mejor aprovechamiento CPU.
- Servicio interactivo a varios usuarios.
- Permite usar soluciones de diseño concurrentes.
- Sincronización y comunicación mediante variables compartidas.

Concurrencia en multiprocesadores de memoria compartida



- Los procesadores pueden compartir o no físicamente la misma memoria, pero comparten un espacio de direcciones compartido.
- La interacción entre los procesos se puede implementar mediante variables alojadas en direccciones del espacio compartido (variables compartidas).
- Ejemplo: PCs con procesadores muticore.

Concurrencia en sistemas distribuidos



- No existe un memoria común: cada procesador tiene su espacio de direcciones privado.
- Los procesadores interaccionan transfiriéndose datos a través de una red de interconexión (paso de mensajes).
- **Programación distribuida**: además de la concurrencia, trata con otros problemas como el tratamiento de los fallos, transparencia, heterogeneidad, etc.
- Ejemplos: Clusters de ordenadores, internet, intranet.

2.2.2 Modelo Abstracto de concurrencia

Sentencias atómicas y no atómicas

Sentencia atómica (indivisible)

Una sentencia o instrucción de un proceso en un programa concurrente es **atómica** si siempre se ejecuta de principio a fin sin verse *afectada* (durante su ejecución) por otras sentencias en ejecución de otros procesos del programa.

- No se verá afectada cuando el *funcionamiento* de dicha instrucción **no dependa nunca** de como se estén ejecutando otras instrucciones.
- El funcionamiento de una instrucción se define por su efecto en el *estado de ejecución* del programa justo cuando acaba.
- El estado de ejecución esta formado por los valores de las variables y los registros de todos los procesos.

Ejemplos de sentencias atómicas.

A modo de ejemplo de instrucciones atómicas, cabe citar muchas de las instrucciones máquina del repertorio de un procesador, por ejemplo estas tres:

- Leer una celda de memoria y cargar su valor en ese momento en un registro del procesador
- Incrementar el valor de un registro (u otras operaciones aritméticas entre registros).
- Escribir el valor de un registro en una celda de memoria.

El resultado de estas instrucciones **no depende nunca** de otras intrucciones que se estén ejecutando concurrentemente. Al finalizar, la celda de memoria o el registro tomará un valor concreto predecible siempre a partir del estado al inicio.

- En el caso de la escritura en memoria, por ejemplo, el hardware asegura que el valor escrito(justo al final de la ejecución) es siempre el que había en el registro (justo al inicio de la ejecución).

Ejemplos de sentencias no atómicas.

La mayoría de las sentencias en lenguajes de alto nivel son típicamente no atómicas, por ejemplo, la sentencia

```
x := x + 1 ; { incrementa el valor de la variable entera 'x' (en RAM) en una unidad }
```

Para ejecutarla , el compilador o intérprete podría usar una secuencia de tres sentencias como esta:

1. leer el valor de `x` y cargarlo en un registro `r` del procesador
2. incrementar en un unidad el valor almacenado en el registro `r`
3. escribir el valor del registro `r` en la variable `x`

El resultado (es decir, el valor que toma `x` justo al acabar) **depende** de que haya o no haya otras sentencias ejecutándose a la vez y escribiendo simultáneamente sobre la variable `x`. Podría ocurrir que el valor al final no sea igual al valor al empezar más uno.

Interfoliación de sentencias atómicas

Supongamos que definimos un programa concurrente C compuesto de dos procesos secuenciales P_A y P_B que se ejecutan a la vez.

La ejecución de C puede dar lugar a cualquiera de las posibles mezclas (**interfoliaciones**) de sentencias atómicas de P_A y P_B.

Pr.	Posibles secuencias de instr. atómicas
P_A	$A_1\ A_2\ A_3\ A_4\ A_5$
P_B	$B_1\ B_2\ B_3\ B_4\ B_5$
C	$A_1\ A_2\ A_3\ A_4\ A_5\ B_1\ B_2\ B_3\ B_4\ B_5$
C	$B_1\ B_2\ B_3\ B_4\ B_5\ A_1\ A_2\ A_3\ A_4\ A_5$
C	$A_1\ B_1\ A_2\ B_2\ A_3\ B_3\ A_4\ B_4\ A_5\ B_5$
C	$B_1\ B_2\ A_1\ B_3\ B_4\ A_2\ B_5\ A_3\ A_4\ A_5$
C	...

las sentencias atómicas se ordenan en función del instante en el que acaban (que es cuando tienen efecto)

Abstracción

El modelo basado en el estudio de todas las posibles secuencias de ejecución entrelazadas de los procesos constituye una **abstracción** sobre las circunstancias de la ejecución de los programas concurrentes, ya que:

- Se consideran exclusivamente las **características relevantes** que determinan el resultado del cálculo
- Esto permite **simplificar** el análisis o creación de los programas concurrentes.

Se **ignoran los detalles no relevantes** para el resultado, como por ejemplo:

- las áreas de memoria asignadas a los procesos
- los registros particulares que están usando
- el costo de los cambios de contexto entre procesos
- la política del S.O. relativa a asignación de CPU
- las diferencias entre entornos multiprocesador o monoprocesador
- ....

Independencia del entorno de ejecución

El entrelazamiento preserva la consistencia

El resultado de una instrucción individual sobre un dato no depende de las circunstancias de la ejecución.

Supongamos que un programa P se compone de dos instrucciones atómicas, I_0 e I_1, que se ejecutan concurrentemente, $P : I_0||I_1$, entonces:

- Si I_0 e I_1 no acceden a la misma celda de memoria o registro, el orden de ejecución no afecta al resultado final.
- Si $I_0 \equiv M \leftarrow 1$ y $I_1 \equiv M \leftarrow 2$, la única suposición razonable es que el resultado sea consistente. Por tanto, al final $M = 1$ ó $M = 2$, pero nunca por ejemplo $M = 3$.

En caso contrario, sería imposible razonar acerca de la corrección de los programas concurrentes.

Velocidad de ejecución de los procesos. Hipótesis del progreso finito

Progreso Finito

No se puede hacer ninguna suposición acerca de las velocidades absolutas/relativas de ejecución de los procesos, salvo que es mayor que cero.

Un programa concurrente se entiende en base a sus componentes (procesos) y sus interacciones, sin tener en cuenta el entorno de ejecución.

Ejemplo: Un disco es más lento que una CPU pero el programa no debería asumir eso en el diseño del programa.

Si se hicieran suposiciones temporales:

- Sería difícil detectar y corregir fallos
- La corrección dependería de la configuración de ejecución, que puede cambiar

Hipótesis del progreso finito

Si se cumple la hipótesis, la velocidad de ejecución de cada proceso será no nula, lo cual tiene estas dos consecuencias:

Punto de vista global

Durante la ejecución de un programa concurrente, en cualquier momento existirá al menos 1 proceso preparado, es decir, eventualmente se permitirá la ejecución de algún proceso.

Punto de vista local

Cuando un proceso concreto de un programa concurrente comienza la ejecución de una sentencia, completará la ejecución de la sentencia en un intervalo de tiempo finito.

Estados e historias de ejecución de un programa concurrente

Estado de un programa concurrente

Valores de las variables del programa en un momento dado. Incluyen variables declaradas explícitamente y variables con información de estado oculta (contador del programa, registros,...).

Un programa concurrente comienza su ejecución en un estado inicial y los procesos van modificando el estado conforme se van ejecutando sus sentencias atómicas (producen transiciones entre dos estados de forma indivisible).

Historia o traza de un programa concurrente

Secuencia de estados $s_0 \to s_1 \to \ldots \to s_n$, producida por una secuencia concreta de interfoliación.

Notaciones para expresar ejecución concurrente

- **Propuestas Iniciales**: no separan la definición de los procesos de su sincronización.
- **Posteriores Propuestas**: separan conceptos e imponen estructura.
- **Declaración de procesos**: rutinas específicas de programación concurrente $\Longrightarrow$ Estructura del programa concurrente más clara.

Sistemas Estáticos

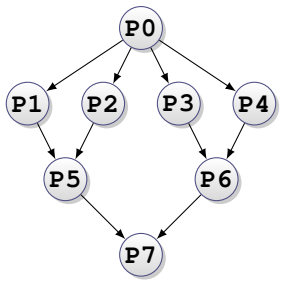
- Número de procesos fijado en el fuente del programa.
- Los procesos se activan al lanzar el programa
- Ejemplo: Message Passing Interface (MPI-1).

Sistemas Dinámicos

- Número variable de procesos/hebras que se pueden activar en cualquier momento de la ejecución.
- Ejemplos: OpenMP, PThreads, Java Threads, MPI-2.

Grafo de Sincronización

- Es un Grafo Dirigido Acíclico (DAG) donde cada nodo representa una secuencia de sentencias del programa (actividad). Dadas dos actividades, A y B, una arista conectando A en dirección hacia B significa que B no puede comenzar su ejecución hasta que A haya finalizado.



- Muestra las restricciones de precedencia que determinan cuándo una actividad puede empezar en un programa.
- Tiene que ser acíclico.

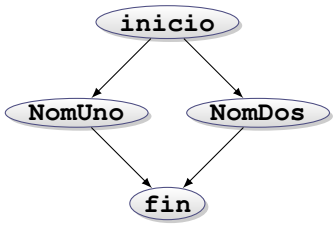
Definición estática de procesos

El número de procesos (arbitrario) y el código que ejecutan no cambian entre ejecuciones. Cada proceso se asocia con su identificador y su código mediante la palabra clave **process**

```
var ....        { vars. compartidas }

process NomUno ;
var  ....       { vars. locales }
begin
   ....         { codigo }
end

process NomDos ;
var ....        { vars. locales }
begin
   ....         { codigo }
end
...             { otros procesos }
```



el programa acaba cuando acaban todos los procesos. Las vars. compartidas se iniciaizan antes de comenzar ningún proceso.

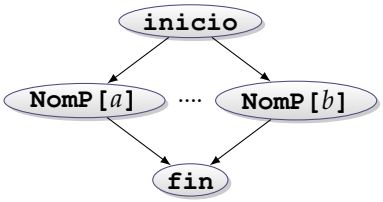
Definición estática de vectores de procesos

Se pueden usar definiciones estáticas de grupos de procesos similares que solo se diferencia en el valor de una constante (**vectores de procesos**)

```
var ....        { vars. compartidas }

process NomP[ ind : a..b ] ;
var  ....       { vars. locales }
begin
   ....         { codigo }
   ....         { aqui ind vale a, a+1,...,b }
end

...             { otros procesos }
```

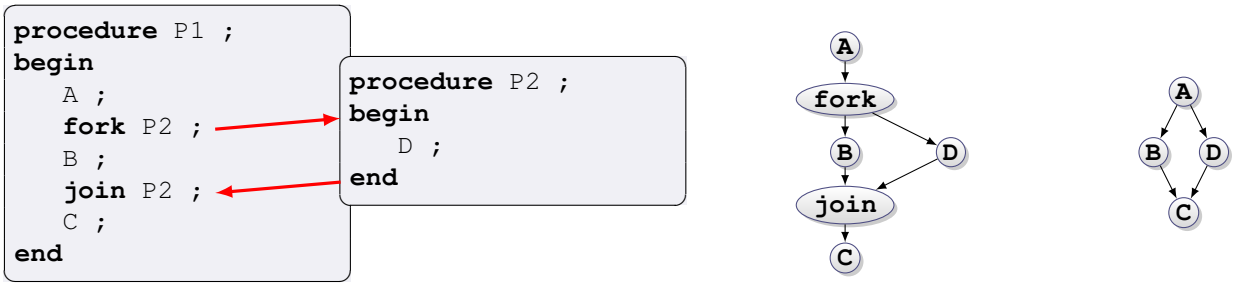


- En cada caso, a y b se traducen por dos constantes concretas (el valor de a será típicamente 0 o 1).
- El número total de procesos será $b-a+1$ (se supone que $a <= b$)

Creación de procesos no estructurada con fork-join.

- **fork**: sentencia que especifica que la rutina nombrada puede comenzar su ejecución, al mismo tiempo que comienza la sentencia siguiente (*bifurcación*).

- **join**: sentencia que espera la terminación de la rutina nombrada, antes de comenzar la sentencia siguiente (*unión*).

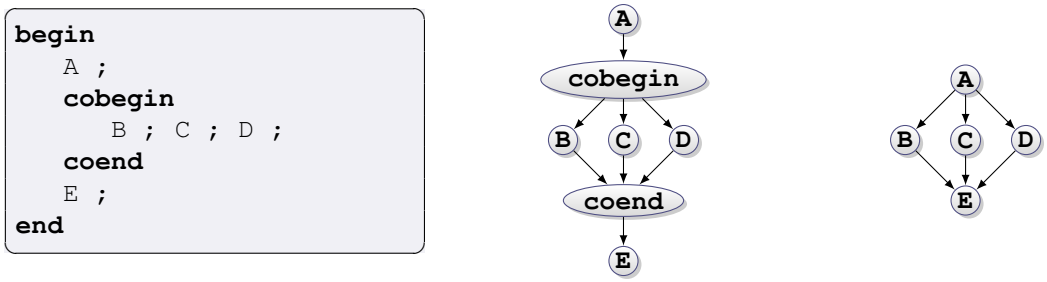


- **Ventajas:** práctica y potente, creación dinámica.
- **Inconvenientes:** no estructuración, difícil comprensión de los programas.

Creación de procesos estructurada con cobegin-coend

Las sentencias en un bloque delimitado por **cobegin–coend** comienzan su ejecución todas ellas a la vez:

- en el **coend** se espera a que se terminen todas las sentencias.
- Hace explícito qué rutinas van a ejecutarse concurrentemente.



- **Ventajas**: impone estructura: 1 única entrada y 1 única salida $\Longrightarrow$ más fácil de entender.
- **Inconveniente**: menor potencia expresiva que **fork-join**.

2.3 Exclusión mutua y sincronización

Exclusión mutua y sincronización

Según el modelo abstracto, los procesos concurrentes ejecutan sus instrucciones atómicas de forma que, en principio, es completamente arbitrario el entremezclado en el tiempo de sus respectivas secuencias de instrucciones. Sin embargo, en un conjunto de procesos que **no son independientes entre sí** (es decir, son **cooperativos**), algunas de las posibles formas de combinar las secuencias no son válidas.

- en general, se dice que hay una **condición de sincronización** cuando esto ocurre, es decir, que hay alguna restricción sobre el orden en el que se pueden mezclar las instrucciones de distintos procesos.

- un caso particular es la **exclusión mutua**, son secuencias finitas de instrucciones que un proceso debe ejecutar de principio a fin sin mezclarse con otras (o las mismas) de otros procesos.

2.3.1 Concepto de exclusión mutua

Exclusión mutua

La restricción se refiere a una o varias secuencias de instrucciones consecutivas que aparecen en el texto de uno o varios procesos.

- Al conjunto de dichas secuencias de instrucciones se le denomina **sección crítica** (SC).
- Ocurre **exclusión mutua** (EM) cuando los procesos solo funcionan correctamente si, en cada instante de tiempo, **hay como mucho uno de ellos ejecutando cualquier instrucción de la sección crítica**.

es decir, el solapamiento de las instrucciones debe ser tal que cada secuencia de instrucciones de la SC se ejecuta como mucho por un proceso de principio a fin, sin que (durante ese tiempo) otros procesos ejecuten ninguna de esas instrucciones ni otras de la misma SC.

Ejemplos de exclusión mutua

El ejemplo típico de EM ocurre en procesos con memoria compartida que acceden para leer y modificar variables o estructuras de datos comunes usando operaciones no atómicas (es decir, compuestas de varias instrucciones máquina o elementales que pueden solaparse con otras secuencias), aunque hay muchos otros ejemplos:

- envío de datos a dispositivos que no se pueden compartir (p.ej., el bucle que envía una secuencia de datos que forma un texto a una impresora o cualquier otro dispositivo de salida vía el bus del sistema).
- recepción de datos desde dispositivos (p.ej., un bucle que lee una secuencia de pulsaciones de teclas desde el teclado, también a través del bus).

Un ejemplo sencillo de exclusión mutua

Para ilustrar el problema de la EM, veremos un ejemplo sencillo que usa una variable entera (x) en memoria compartida y operaciones aritméticas elementales.

- La sección crítica esta formada por todas las secuencias de instrucciones máquina que se obtienen al traducir (compilar) operaciones de escritura (o lectura y escritura) de la variable (p.ej., asignaciones como $x:=x+1$ o $x:=4*z$).
- Veremos que si varios procesos ejecutan las instrucciones máquina de la sección crítica de forma simultánea, los resultados de este tipo de asignaciones son **indeterminados**.

aquí, el término *indeterminado* indica que para cada valor de x (o del resto de variables) previo a cada asignación, existe un conjunto de valores distintos posibles de x al finalizar la ejecución de dicha asignación (el valor concreto que toma x depende del orden de entremezclado de las instrucciones máquina).

Traducción y ejecución de asignaciones

Si consideramos la instrucción **x:=x+1** (que forma la SC), veremos que una traducción típica a código máquina tendría estas tres instrucciones:

1.	**load** $r_i \leftarrow x$	cargar el valor de la variable **x** en un registro **r** de la CPU (por el proceso número i).
2.	**add** r_i **,1**	incrementar en una unidad el valor del registro **r**
3.	**store** $r_i \rightarrow x$	guardar el valor del registro **r** en la posición de memoria de la variable **x**.

- hay dos procesos concurrentes (P_0 y P_1) que ejecutan la asignación, y por tanto las tres instrucciones máquina se pueden entremezclar de forma arbitraria.
- cada proceso mantiene su propia copia del registro **r** (los llamaremos r_0 y r_1)
- ambos comparten **x**, cuyos accesos vía **load** y **store** son atómicos pues bloquean el bus del sistema.

Posibles secuencias de instrucciones

Suponemos que inicialmente **x** vale 0 y ambos procesos ejecutan la asignación, puede haber varias secuencias de interfoliación, aquí vemos dos:

P_0	P_1	x
load $r_0 \leftarrow x$		0
add r_0 **,1**		0
store $r_0 \rightarrow x$		1
	load $r_1 \leftarrow x$	1
	add r_1 **,1**	1
	store $r_1 \rightarrow x$	2

P_0	P_1	x
load $r_0 \leftarrow x$		0
	load $r_1 \leftarrow x$	0
add r_0 **,1**		0
	add r_1 **,1**	0
store $r_0 \rightarrow x$		1
	store $r_1 \rightarrow x$	1

por tanto, partiendo de x== 0, tenemos al final que la variable puede tomar el valor 1 o 2 dependiendo del orden de ejecución de las instrucciones.

2.3.2 Condición de sincronización

Condición de sincronización.

En general, en un programa concurrente compuesto de varios procesos, una **condición de sincronización** establece que:

> no son correctas todas las posibles interfoliaciones de las secuencias de instrucciones atómicas de los procesos.

- esto ocurre típicamente cuando, en un punto concreto de su ejecución, uno o varios procesos deben esperar a que se cumpla una determinada condición global (depende de varios procesos).

Veremos un ejemplo sencillo de condición de sincronización en el caso en que los procesos puedan usar variables comunes para comunicarse (memoria compartida). En este caso, los accesos a las variables no pueden ordenarse arbitrariamente (p.ej.: leer de ella antes de que sea escrita)

Ejemplo de sincronización. Productor Consumidor

Un ejemplo típico es el de dos procesos cooperantes en los cuales uno de ellos (productor) produce una secuencia de valores (p.ej. enteros) y el otro (consumidor) usa cada uno de esos valores. La comunicación se hace vía la variable compartida x.:

```
{ variables compartidas }
var x : integer ; { contiene cada valor producido }
```

```
{ Proceso productor: calcula 'x' }
process Productor ;
   var a : integer ; { no compartida }
begin
   while true do begin
      { calcular un valor }
      a := ProducirValor() ;
      { escribir en mem. compartida }
      x := a ; { sentencia E }
   end
end
```

```
{ Proceso Consumidor: lee 'x' }
process Consumidor ;
   var b : integer ; { no compartida }
begin
   while true do begin
      { leer de mem. compartida }
      b := x ; { sentencia L }
      { utilizar el valor leido }
      UsarValor(b) ;
   end
end
```

Secuencias correctas e incorrectas

Los procesos descritos solo funcionan como se espera si el orden en el que se entremezclan las sentencias elementales etiquetadas como E (escritura) y L (lectura) es: $E, L, E, L, E, L, \ldots$.

- $L, E, L, E, \ldots$ es incorrecta: se hace una lectura de x previa a cualquier escritura (se lee valor indeterminado).
- $E, L, E, E, L, \ldots$ es incorrecta: hay dos escrituras sin ninguna lectura entre ellas (se produce un valor que no se lee).
- $E, L, L, E, L, \ldots$ es incorrecta: hay dos lecturas de un mismo valor, que por tanto es usado dos veces.

La secuencia válida asegura la condición de sincronización:

- Consumidor no lee hasta que Productor escriba un nuevo valor en x (cada valor producido es usado una sola vez).
- Productor no escribe un nuevo valor hasta que Consumidor lea el último valor almacenado en x (ningún valor producido se pierde).

2.4 Propiedades de los sistemas concurrentes

2.4.1 Corrección de un sistema concurrente

Concepto de corrección de un programa concurrente

Propiedad de un programa concurrente: Atributo del programa que es cierto para todas las posibles secuencias de interfoliación (historias del programa).
Hay 2 tipos:

- **Propiedad de seguridad** (*safety*).
- **Propiedad de vivacidad** (*liveness*).

2.4.2 Propiedades de seguridad y vivacidad

Propiedades de Seguridad (Safety)

- Son condiciones que *deben cumplirse siempre* del tipo: *Nunca pasará nada malo*.
- Requeridas en especificaciones estáticas del programa.
- Similar a *corrección parcial* en programas secuenciales: "Si el programa termina, las respuestas deben ser correctas.
- Son fáciles de demostrar y para cumplirlas se suelen restringir las posibles interfoliaciones.

Ejemplos:

- **Exclusión mutua**: 2 procesos nunca entrelazan ciertas subsecuencias de operaciones.
- **Ausencia Interbloqueo (Deadlock-freedom)**: Nunca ocurrirá que los procesos se encuentren esperando algo que nunca sucederá.
- **Propiedad de seguridad en el Productor-Consumidor** El consumidor debe consumir todos los datos producidos por el productor en el orden en que se van produciendo.

Propiedades de Vivacidad (Liveness)

- *Deben cumplirse eventualmente*.
- Son propiedades dinámicas difíciles de probar: *Realmente sucede algo bueno*

Ejemplos:

- **Ausencia de inanición (starvation-freedom)**: Un proceso o grupo de procesos no puede ser indefinidamente pospuesto. En algún momento, podrá avanzar.
- **Equidad (fairness)**: Tipo particular de prop. de vivacidad. Un proceso que desee progresar debe hacerlo con justicia relativa con respecto a los demás. Más ligado a la implementación y a veces incumplida: existen distintos grados.

2.5 Verificación de programas concurrentes

2.5.1 Verificación de programas concurrentes. Introducción

Verificación de programas concurrentes. Introducción

¿ Cómo demostrar que un programa cumple una determinada propiedad ?

- **Posibilidad:** realizar diferentes ejecuciones del programa y comprobar que se verifica la propiedad.
- **Problema**: Sólo permite considerar un número limitado de historias de ejecución y no demuestra que no existan casos indeseables.
- **Ejemplo**: Comprobar que el proceso P produce al final $x = 3$:

```
process P ;
   var x : integer := 0 ;
cobegin
   x := x+1 ; x := x+2 ;
coend
```

  Hay varias historias que llevan a `x==1` o `x==2` (la asignación no es atómica), pero estas historias podrían no aparecer dentro de un número limitado de ejecuciones.

Verificación de programas concurrentes. Enfoque operacional

- **Enfoque operacional**: Análisis exhaustivo de casos. Se chequea la corrección de todas las posibles historias.
- **Problema**: Su utilidad está muy limitada cuando se aplica a programas concurrentes complejos ya que el número de interfoliaciones crece exponencialmente con el número de instrucciones de los procesos.
- Para el sencillo programa P (2 procesos, 3 sentencias atómicas por proceso) habría que estudiar 20 historias disferentes.

2.5.2 Enfoque axiomático para la verificación

Verificación. Enfoque axiomático

- Se define un *sistema lógico formal* que permite establecer propiedades de programas en base a axiomas y reglas de inferencia.
- Se usan fórmulas lógicas (asertos) para caracterizar un conjunto de estados.

- Las sentencias atómicas actúan como *transformadores de predicados* (asertos). Los teoremas en la lógica tienen la forma:

$$\{P\} \quad \mathbf{S} \quad \{Q\}$$

"Si la ejecución de la sentencia S empieza en algún estado que hace verdadero el predicado P (*precondición*), entonces el predicado Q (*poscondición*) será verdadero en el estado resultante.

- **Menor Complejidad**: El trabajo que conlleva la prueba de corrección es proporcional al número de sentencias atómicas en el programa.

Invariante global

- **Invariante global**: Predicado que referencia variables globales siendo cierto en el estado inicial de cada proceso y manteniéndose cierto ante cualquier asignación dentro de los procesos.
- En una solución correcta del Productor-Consumidor, un invariante global sería:

$$consumidos \leq producidos \leq consumidos + 1$$

Bibliografía del tema 1.

Para más información, ejercicios, bibliografía adicional, se puede consultar:

1.1. **Conceptos básicos y Motivación** Palma (2003), capítulo 1.

1.2. **Modelo abstracto y Consideraciones sobre el hardware** Ben-Ari (2006), capítulo 2. Andrews (2000) capítulo 1. Palma (2003) capítulo 1.

1.3. **Exclusión mutua y sincronización** Palma (2003), capítulo 1.

1.4. **Propiedades de los Sistemas Concurrentes** Palma (2003), capítulo 1.

1.5. **Verificación de Programas concurrentes** Andrews (2000), capítulo 2.

2.6 Problemas del tema 1.

1

Considerar el siguiente fragmento de programa para 2 procesos P_1 y P_2:

Los dos procesos pueden ejecutarse a cualquier velocidad. ¿ Cuáles son los posibles valores resultantes para x ?. Suponer que x debe ser cargada en un registro para incrementarse y que cada proceso usa un registro diferente para realizar el incremento.

```
{ variables compartidas }
var x : integer := 0 ;
```

```
process P1 ;
   var i : integer ;
begin
   for i := 1 to 2 do begin
      x := x+1 ;
   end
end
```

```
process P2 ;
   var j : integer ;
begin
   for j := 1 to 2 do begin
      x := x+1 ;
   end
end
```

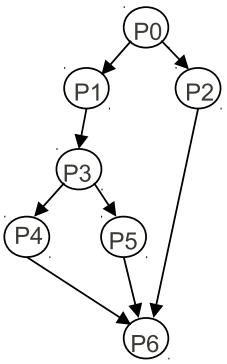
2

¿ Cómo se podría hacer la copia del fichero `f` en otro `g`, de forma concurrente, utilizando la instrucción concurrente **cobegin–coend** ? . Para ello, suponer que:

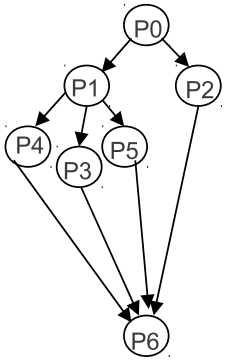
- los archivos son secuencia de items de un tipo arbitrario `T`, y se encuentran ya abiertos para lectura (`f`) y escritura (`g`). Para leer un ítem de `f` se usa la llamada a función **leer**`(f)` y para saber si se han leído todos los ítems de `f`, se puede usar la llamada **fin**`(f)` que devuelve verdadero si ha habido al menos un intento de leer cuando ya no quedan datos. Para escribir un dato x en `g` se puede usar la llamada a procedimiento **escribir**`(g,`x`)`.
- El orden de los ítems escritos en `g` debe coincidir con el de `f`.
- Dos accesos a dos archivos distintos pueden solaparse en el tiempo.

3

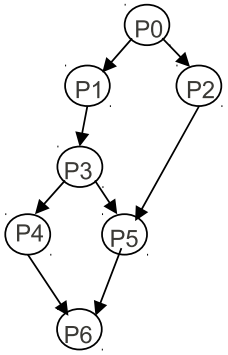
Construir, utilizando las instrucciones concurrentes **cobegin–coend** y **fork–join**, programas concurrentes que se correspondan con los grafos de precedencia que se muestran a continuación:



a)



b)



c)

4

Dados los siguientes fragmentos de programas concurrentes, obtener sus grafos de precedencia asociados:

```
begin
   P0 ;
   cobegin
      P1 ;
      P2 ;
      cobegin
         P3 ; P4 ; P5 ; P6 ;
      coend
      P7 ;
   coend
   P8;
end
```

```
begin
   P0 ;
   cobegin
      begin
        cobegin
          P1;P2;
        coend
       P5;
      end
      begin
        cobegin
          P3;P4;
        coend
       P6;
      end
   coend
   P7 ;
end
```

5

Suponer un sistema de tiempo real que dispone de un captador de impulsos conectado a un contador de energía eléctrica. La función del sistema consiste en contar el número de impulsos producidos en 1 hora (cada *Kwh* consumido se cuenta como un impulso) e imprimir este número en un dispositivo de salida.

Para ello se dispone de un programa concurrente con 2 procesos: un proceso acumulador (lleva la cuenta de los impulsos recibidos) y un proceso escritor (escribe en la impresora). En la variable común a los 2 procesos `n` se lleva la cuenta de los impulsos. El proceso acumulador puede invocar un procedimiento `Espera_impulso` para esperar a que llegue un impulso, y el proceso escritor puede llamar a `Espera_fin_hora` para esperar a que termine una hora.

El código de los procesos de este programa podría ser el siguiente:

```
{ variable compartida: }
var n : integer; { contabiliza impulsos }
```

```
process Acumulador ;
begin
   while true do begin
      Espera_impulso();
      < n := n+1 > ; { (1) }
   end
 end
```

```
process Escritor ;
begin
   while true do begin
      Espera_fin_hora();
      write( n ) ;    { (2) }
      < n := 0 > ;    { (3) }
   end
end
```

En el programa se usan sentencias de acceso a la variable `n` encerradas entre los símbolos `<` y `>`. Esto significa que cada una de esas sentencias se ejecuta en exclusión mutua entre los dos procesos, es decir, esas sentencias se ejecutan de principio a fin sin entremezclarse entre ellas.

Supongamos que en un instante dado el acumulador está esperando un impulso, el escritor está esperando el fin de una hora, y la variable `n` vale k. Después se produce de forma simultánea un nuevo impulso y el fin del periódo de una hora. Obtener las posibles secuencias de interfolicación de las instrucciones (1),(2), y (3) a partir de dicho instante, e indicar cuales de ellas son correctas y cuales incorrectas (las incorrectas son aquellas en las cuales el impulso no se contabiliza).

6

Supongamos un programa concurrente en el cual hay, en memoria compartida dos vectores `a` y `b` de enteros y con tamaño par, declarados como sigue:

```
var a,b : array[1..2*n] of integer ; { n es una constante predefinida }
```

Queremos escribir un programa para obtener en `b` una copia ordenada del contenido de `a` (nos da igual el estado en que queda `a` después de obtener `b`).

Para ello disponemos de la función **Sort** que ordena un tramo de `a` (entre las entradas `s` y `t`, ambas incluidas). También disponemos la función `Copiar`, que copia un tramo de `a` (desde `s` hasta `t`) en `b` (a partir de `o`)

```
procedure Sort( s,t : integer );
   var i, j : integer ;
begin
  for i := s to t do
     for j:= s+1 to t do
        if a[i] < a[j] then
           swap( a[i], b[j] ) ;
end
```

```
procedure Copiar( o,s,t : integer );
   var d : integer ;
begin
   for d := 0 to t-s do
      b[o+d] := a[s+d] ;
end
```

El programa para ordenar se puede implementar de dos formas:

- Ordenar todo el vector `a`, de forma secuencial con la función **Sort**, y después copiar cada entrada de `a` en `b`, con la función `Copiar`.

- Ordenar las dos mitades de `a` de forma concurrente, y después mezclar dichas dos mitades en un segundo vector `b` (para mezclar usamos un procedimiento **`Merge`**).

A continuación vemos el código de ambas versiones:

```
procedure Secuencial() ;
   var i : integer ;
begin
   Sort( 1, 2*n );    { ordena a    }
   Copiar( 1, 2*n ); { copia a en b }
end
```

```
procedure Concurrente() ;
begin
  cobegin
     Sort( 1,   n   );
     Sort( n+1, 2*n );
  coend
  Merge( 1, n+1, 2*n );
end
```

El código de **`Merge`** se encarga de ir leyendo las dos mitades de `a`, en cada paso, seleccionar el menor elemento de los dos siguientes por leer (uno en cada mitad), y escribir dicho menor elemento en la siguiente mitad del vector mezclado `b`. El código es el siguiente:

```
procedure Merge( inferior, medio, superior: integer ) ;
   var escribir : integer := 1 ;         { siguiente posicion a escribir en b       }
   var leer1    : integer := inferior ; { siguiente pos. a leer en primera mitad de a }
   var leer2    : integer := medio    ; { siguiente pos. a leer en segunda mitad de a }
begin
   { mientras no haya terminado con alguna mitad }
   while leer1 < medio and leer2 <= superior do begin
      if a[leer1] < a[leer2] then begin { minimo en la primera mitad }
         b[escribir] := a[leer1] ;
         leer1 := leer1 + 1 ;
      end else begin { minimo en la segunda mitad }
         b[escribir] := a[leer2] ;
         leer2 := leer2 + 1 ;
      end
      escribir := escribir+1 ;
   end
   { se ha terminado de copiar una de las mitades, copiar lo que quede de la otra }
   if leer2 > superior then Copiar( escribir, leer1, medio-1   ); { copiar primera }
                       else Copiar( escribir, leer2, superior ); { copiar segunda }
end
```

Llamaremos $T_s(k)$ al tiempo que tarda el procedimiento **`Sort`** cuando actua sobre un segmento del vector con k entradas. Suponemos que el tiempo que (en media) tarda cada iteración del bucle interno que hay en **`Sort`** es la unidad (por definición). Es evidente que ese bucle tiene $k(k-1)/2$ iteraciones, luego:

$$T_s(k) = \frac{k(k-1)}{2} = \frac{1}{2}k^2 - \frac{1}{2}k$$

El tiempo que tarda la versión secuencial sobre $2n$ elementos (llamaremos S a dicho tiempo) será evidentemente $T_s(2n)$, luego

$$S = T_s(2n) = \frac{1}{2}(2n)^2 - \frac{1}{2}(2n) = 2n^2 - n$$

con estas definiciones, calcula el tiempo que tardará la versión paralela, en dos casos:

(1) Las dos instancias concurrentes de **Sort** se ejecutan en el mismo procesador (llamamos P_1 al tiempo que tarda).

(2) Cada instancia de **Sort** se ejecuta en un procesador distinto (lo llamamos P_2)

escribe una comparación cualitativa de los tres tiempos (S,P_1 y P_2).

Para esto, hay que suponer que cuando el procedimiento **Merge** actua sobre un vector con p entradas, tarda p unidades de tiempo en ello, lo cual es razonable teniendo en cuenta que en esas circunstancias **Merge** copia p valores desde `a` hacia `b`. Si llamamos a este tiempo $T_m(p)$, podemos escribir

$$T_m(p) \quad = \quad p$$

.

7

Supongamos que tenemos un programa con tres matrices (`a`,`b` y `c`) de valores flotantes declaradas como variables globales. La multiplicación secuencial de `a` y `b` (almacenando el resultado en `c`) se puede hacer mediante un procedimiento **MultiplicacionSec** declarado como aparece aquí:

```
var a, b, c : array[1..3,1..3] of real ;

procedure MultiplicacionSec()
   var i,j,k : integer ;
begin
   for i := 1 to 3 do
      for j := 1 to 3 do begin
         c[i,j] := 0 ;
         for k := 1 to 3 do
            c[i,j] := c[i,j] + a[i,k]*b[k,j] ;
      end
end
```

Escribir un programa con el mismo fin, pero que use 3 procesos concurrentes. Suponer que los elementos de las matrices `a` y `b` se pueden leer simultáneamente, así como que elementos distintos de `c` pueden escribirse simultáneamente.

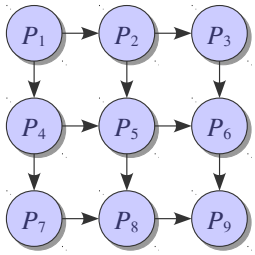
8

Un trozo de programa ejecuta nueve rutinas o actividades ($P_1, P_2, \ldots, P_9$), repetidas veces, de forma concurrentemente con **cobegin coend** (ver la figura de la izquierda), pero que requieren sincronizarse según determinado grafo (ver la figura de la derecha):

Trozo de programa:

```
while true do
cobegin
    P1 ; P2 ; P3 ;
    P4 ; P5 ; P6 ;
    P7 ; P8 ; P9 ;
coend
```

Grafo de sincronización:



Supón que queremos realizar la sincronización indicada en el grafo, usando para ello llamadas desde cada rutina a dos procedimientos (**`EsperarPor`** y **`Acabar`**). Se dan los siguientes hechos:

- El procedimiento **`EsperarPor`**`(i)` es llamado por una rutina cualquiera (la número `k`) para esperar a que termine la rutina número `i`, usando espera ocupada. Por tanto, se usa por la rutina `k` al inicio para esperar la terminación de las otras rutinas que corresponda según el grafo.
- El procedimiento **`Acabar`**`(i)` es llamado por la rutina número `i`, al final de la misma, para indicar que dicha rutina ya ha finalizado.
- Ambos procedimientos pueden acceder a variables globales en memoria compartida.
- Las rutinas se sincronizan única y exclusivamente mediante llamadas a estos procedimientos, siendo la implementación de los mismos completamente transparente para las rutinas.

Escribe una implementación de **`EsperarPor`** y **`Acabar`** (junto con la declaración e inicialización de las variables compartidas necesarias) que cumpla con los requisitos dados.

Chapter 3

Tema 2. Sincronización en memoria compartida.

3.1 Introducción a la sincronización en memoria compartida.

Sincronización en memoria compartida

En esta tema estudiaremos soluciones para exclusión mutua y sincronización basadas en el uso de memoria compartida entre los procesos involucrados. Este tipo de soluciones se pueden dividir en dos categorías:

- **Soluciones de bajo nivel con espera ocupada** están basadas en programas que contienen explícitamente instrucciones de bajo nivel para lectura y escritura directamente a la memoria compartida, y bucles para realizar las esperas.
- **Soluciones de alto nivel** partiendo de las anteriores, se diseña una capa software por encima que ofrece un interfaz para las aplicaciones. La sincronización se consigue bloqueando un proceso cuando deba esperar.

Soluciones de bajo nivel con espera ocupada

Cuando un proceso debe esperar a que ocurra un evento o sea cierta determinada condición, entra en un bucle indefinido en el cual continuamente comprueba si la situación ya se da o no (a esto se le llama **espera ocupada**). Este tipo de soluciones se pueden dividir en dos categorías:

- **Soluciones software**: se usan operaciones estándar sencillas de lectura y escritura de datos simples (típicamente valores lógicos o enteros) en la memoria compartida
- **Soluciones hardware** (**cerrojos**): basadas en la existencia de instrucciones máquina específicas dentro del repertorio de instrucciones de los procesadores involucrados

Soluciones de alto nivel

Las soluciones de bajo nivel con espera ocupada se prestan a errores, producen algoritmos complicados y tienen un impacto negativo en la eficiencia de uso de la CPU (por los bucles). En las soluciones de alto

nivel se ofrecen interfaces de acceso a estructuras de datos y además se usa bloqueo de procesos en lugar de espera ocupada. Veremos algunas de estas soluciones:

- **Semáforos**: se construyen directamente sobre las soluciones de bajo nivel, usando además servicios del SO que dan la capacidad de bloquear y reactivar procesos.
- **Regiones críticas condicionales**: son soluciones de más alto nivel que los semáforos, y que se pueden implementar sobre ellos.
- **Monitores:** son soluciones de más alto nivel que las anteriores y se pueden implementar en algunos lenguajes orientados a objetos como Java o Python.

3.2 Semáforos para sincronización

3.2.1 Introducción

Semáforos. Introducción

Los **semáforos** constituyen un mecanismo que soluciona o aminora los problemas de las soluciones de bajo nivel, y tienen un ámbito de uso más amplio:

- no se usa espera ocupada, sino bloqueo de procesos (uso mucho más eficiente de la CPU)
- resuelven fácilmente el problema de exclusión mutua con esquemas de uso sencillos
- se pueden usar para resolver problemas de sincronización (aunque en ocasiones los esquemas de uso son complejos)
- el mecanismo se implementa mediante instancias de una estructura de datos a las que se accede únicamente mediante subprogramas específicos. Esto aumenta la seguridad y simplicidad.

Bloqueo y desbloqueo de procesos

Los semáforos exigen que los procesos que deseen acceder a una SC no ocupen la CPU mientras esperan a que otro proceso o hebra abandone dicha SC, esto implica que:

- un proceso en ejecución debe poder solicitar quedarse bloqueado
- un proceso bloqueado no puede ejecutar instrucciones en la CPU
- un proceso en ejecución debe poder solicitar que se desbloquee (se reanude) algún otro proceso bloqueado.
- deben poder existir simultáneamente varios conjuntos de procesos bloqueados.
- cada petición de bloqueo o desbloqueo se debe referir a alguno de estos conjuntos.

Todo esto requiere el uso de servicios externos (proporcionados por el sistema operativo o por la librería de hebras), mediante una interfaz bien definida.

3.2.2 Estructura de un semáforo

Estructura de un semáforo

Un semáforo es un instancia de una estructura de datos (un registro) que contiene los siguientes elementos:

- Un conjunto de procesos bloqueados (de estos procesos decimos que están esperando al semáforo).
- Un valor natural (un valor entero no negativo), al que llamaremos por simplicidad *valor del semáforo*

Estas estructuras de datos residen en memoria compartida. Al inicio de un programa que los usa debe poder inicializarse cada semáforo:

- el conjunto de procesos asociados estará vacío
- se deberá indicar un valor inicial del semáforo

3.2.3 Operaciones sobre los semáforos.

Operaciones sobre los semáforos

Además de la inicialización, solo hay dos operaciones básicas que se pueden realizar sobre una variable u objeto cualquiera de tipo semáforo (que llamamos s) :

- **sem_wait**(s)
  - Si el valor de s es mayor que cero, decrementar en una unidad dicho valor
  - En otro caso (si el valor de s es cero), bloquear el proceso que invoca en el conjunto de procesos bloqueados asociado a s
- **sem_signal**(s)
  - Si el conjunto de procesos bloqueados asociado a s no está vacío, desbloquear uno de dichos procesos.
  - En otro caso (si el conjunto de procesos bloqueados asociado a s está vacío), incrementar en una unidad el valor de s.

Invariante de un semáforo

Dado un semáforo s, en un instante de tiempo cualquiera t (en el cual el valor de s no esté en proceso de actualización) se cumplirá que:

$$v_0 + n_s = v_t + n_w$$

todos estos símbolos desginan números enteros no negativos, en concreto:

v_0 = valor inicial de s

v_t = valor de s en el instante t.

n_s = número total de llamadas a **sem_signal**(s) completadas hasta el instante t.

n_w = número total de llamadas a **sem_wait**(s) completadas hasta el instante t (no se incluyen los **sem_wait** no completados por haber dejado el proceso en espera).

Implementación de sem_wait y sem_signal.

Se pueden implementar con este esquema:

```
procedure sem_wait(var s:semaphore);
begin
   if s.valor > 0 then
      s.valor := s.valor - 1 ;
   else
      bloquear( s.procesos ) ;
end
```

```
procedure sem_signal(var s:semaphore);
begin
   if vacio( s.procesos ) then
      s.valor := s.valor + 1 ;
   else
      desbloquear( s.procesos ) ;
end
```

En un semáforo cualquiera, estas operaciones se ejecutan de forma atómica, es decir, no puede haber dos procesos distintos ejecutando estas operaciones a la vez sobre un mismo semáforo (excluyendo el período de bloqueo que potencialmente conlleva la llamada a **sem_wait**).

3.2.4 Uso de los semáforos

Uso de semáforos para exclusión mutua

Los semáforos se pueden usar para EM usando un semáforo inicializado a 1, y haciendo **sem_wait** antes de la sección crítica y **sem_signal** después de la sección crítica:

```
{ variables compartidas y valores iniciales }
var sc_libre : semaphore := 1 ; { vale 1 si SC esta libre, 0 si SC esta ocupada }

{ procesos }
process P[ i : 0..n ];
begin
   while true do begin
      sem_wait( sc_libre ) ; { esperar bloqueado hasta que sc_libre sea 1 }
         { seccion critica: ..... }
      sem_signal( sc_libre ) ; { desbloquear proc. en espera o poner sc_libre a 1 }
      { resto seccion: ..... }
   end
end
```

En cualquier instante de tiempo, la suma del valor del semáforo más el número de procesos en la SC es la unidad. Por tanto, solo puede haber 0 o 1 procesos en SC, y se cumple la exclusión mutua.

Uso de semáforos para sincronización

El problema del Productor-Consumidor que vimos se puede resolver fácilmente con semáforos:

```
{ variables compartidas y valores iniciales }
var x              : integer ;          { contiene cada valor producido }
    producidos     : integer :=0 ;      { numero de valores producidos }
    consumidos     : integer :=0 ;      { numero de valores consumidos }
    puede_leer     : semaphore := 0 ;   { 1 si se puede leer 'x', 0 en otro caso }
    puede_escribir : semaphore := 1 ;   { 1 si se puede escribir 'x', 0 en otro caso }
```

```
process Productor ; { calcula 'x' }
var a : integer ;
begin
   while true do begin
      a := ProducirValor() ;
      sem_wait( puede_escribir  );
      x := a ; { sentencia E }
      producidos := producidos+1 ;
      sem_signal( puede_leer ) ;
   end
end
```

```
process Consumidor  { lee 'x' }
var b : integer ;
begin
   while true do begin
      sem_wait( puede_leer ) ;
      b := x ; { sentencia L }
      consumidos := consumidos+1 ;
      sem_signal( puede_escribir ) ;
      UsarValor(b) ;
   end
end
```

Limitaciones de los semáforos

Los semáforos resuelven de una forma eficiente y sencilla el problema de la exclusión mutua y problemas sencillos de sincronización, sin embargo:

- los problemas más complejos de sincronización se resuelven de forma algo más compleja (es difícil verificar su validez, y es fácil que sean incorrectos).
- al igual que los cerrojos, programas erróneos o malintencionados pueden provocar que haya procesos bloqueados indefinidamente o en estados incorrectos.

En la siguiente sección se verá una solución de más alto nivel sin estas limitaciones (monitores).

3.3 Monitores como mecanismo de alto nivel

3.3.1 Fundamento teórico de los monitores

Fundamento teórico de los monitores

- **Inconvenientes de usar mecanismos como los semáforos**:
  - Basados en variables globales: esto impide un diseño modular y reduce la escalabilidad (incorporar más procesos al programa suele requerir la revisión del uso de las variables globales).
  - El uso y función de las variables no se hace explícito en el programa, lo cual dificulta razonar sobre la corrección de los programas.
  - Las operaciones se encuentran dispersas y no protegidas (posibilidad de errores).
- Es necesario un mecanismo que permita el acceso estructurado y la encapsulación, y que además proporcione herramientas para garantizar la exclusión mutua e implementar condiciones de sincronización

3.3.2 Definición de monitor

Definición de monitor

- *C.A.R. Hoare (1974)* **Monitor**: mecanismo de **alto nivel** que permite:
  - Definir **objetos abstractos compartidos**:
    - Colección de variables encapsuladas (datos) que representan un recurso compartido por varios procesos.
    - Conjunto de procedimientos para manipular el recurso: afectan a las variables encapsuladas.
  - Garantizar el **acceso exclusivo** a las variables e implementar **sincronización**.
- El **recurso compartido** se percibe como un objeto al que se accede concurrentemente.
- **Acceso estructurado y Encapsulación**:
  - El usuario (proceso) solo puede acceder al recurso mediante un conjunto de operaciones.
  - El usuario ignora la/s variable/s que representan al recurso y la implementación de las operaciones asociadas.

Propiedades de un monitor

Exclusión mutua en el acceso a los procedimientos

- La exclusión mutua en el acceso a los procedimientos del monitor está garantizada por definición.
- La implementación del monitor garantiza que nunca dos procesos estarán ejecutando simultáneamente algún procedimiento del monitor (el mismo o distintos).

Ventajas sobre los semáforos (solución no estructurada)

- Las variables están protegidas, evitando interferencias exteriores.
- **Acceso estructurado**: Un proceso cualquiera solo puede acceder a las variables del monitor usando los procedimientos exportados por el monitor. Toda la espera y señalización se programan dentro del monitor. Si el monitor es correcto, lo será cada instancia usada por los procesos.
- **Exclusión mutua garantizada**: evita errores.

Sintaxis de un monitor

Estructura de una declaración de un monitor, de nombre *nombre-monitor*, con N procedimientos (con nombres: *nom_proc_1*,*nom_proc_2*,...,*nom_proc_n*), de los cuales algunos son exportados (*nom_exp_1*,*nom_exp_2*,....):

```
monitor nombre_monitor ;

   var                              { declaracion de variables permanentes (privadas)  }
      ..... ;                       {     (se conservan entre llamadas al monitor)  }
   export                           { declaracion de procedimientos publicos     }
     nom_exp_1, nom_exp_2 .... ;     {     (se indican solo los nombres)  }

   procedure nom_proc_1( ... ); { decl. de procedimiento y sus parametros formales }
```

```
      var ...    ;                { declaracion de variables locales a nom_proc_1 }
   begin
      ...                         { codigo que implementa  nom_proc_1 }
   end
   ...                            { resto de procedimientos del monitor }

begin
   ....                           { codigo de inicializacion de vars. permanentes }
end
```

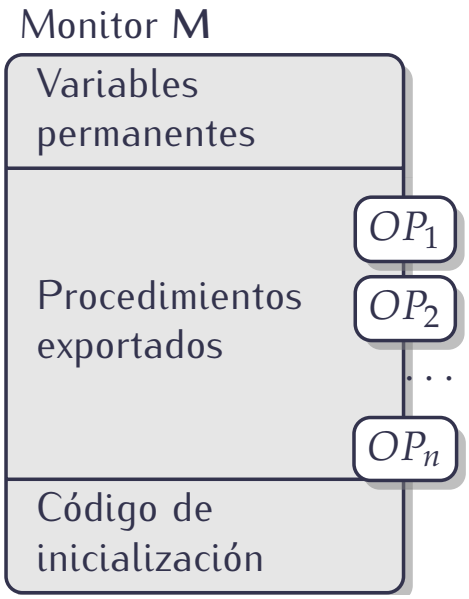
Componentes de un monitor

1. Un conjunto de **variables permanentes**:
   - Almacenan el estado interno del recurso compartido que está siendo representado por el monitor.
   - Sólo pueden ser accedidas dentro del monitor (en el cuerpo de los procedimientos y código de inicialización).
   - Permanecen sin modificaciones entre dos llamadas consecutivas a procedimientos del monitor.
2. Un conjunto de **procedimientos internos**:
   - Manipulan las variables permanentes.
   - Pueden tener variables y parámetros locales, que toman un nuevo valor en cada activación del procedimiento.
   - Algunos (o todos) constituyen la interfaz externa del monitor y podrán ser llamados por los procesos que comparten el recurso.
3. Un **código de inicialización**:
   - Inicializa las variables permanentes del monitor.
   - Se ejecuta una única vez, antes de cualquier llamada a procedimiento del monitor.

Diagrama de las componentes del monitor

El monitor se puede visualizar como aparece en el diagrama:

- El uso que se hace del monitor se hace **exclusivamente** usando los procedimientos exportados (constituyen el interfaz con el exterior).
- Las variables permanentes y los procedimientos no exportados **no son accesibles** desde fuera.
- **Ventaja:** la implementación de las operaciones se puede cambiar sin modificar su semántica.



Ejemplo de monitor (pseudocódigo)

Tenemos varios procesos que pueden incrementar (en una unidad) una variable compartida y poder examinar su valor en cualquier momento:

```
{ declaracion del monitor }

monitor VariableCompartida ;

  var x : integer; { permanente }

  export incremento, valor;

  procedure incremento( );
  begin
       x := x+1 ; { incrementa valor actual }
  end;
  procedure valor(var v:integer);
  begin
       v := x ; { copia sobre v valor actual }
  end;

begin
  x := 0 ; { inicializa valor }
end
```

```
{ ejemplo de uso del monitor        }
{ (debe aparecer en el ambito de la }
{  declaracion del monitor)         }


VariableCompartida.incremento();
VariableCompartida.valor( k ) ;
```

3.3.3 Funcionamiento de los monitores

Funcionamiento de los monitores

- **Comunicación monitor-mundo exterior**: Cuando un proceso necesita operar sobre un recurso compartido controlado por un monitor deberá realizar una llamada a uno de los procedimientos exportados por el monitor usando los parámetros reales apropiados.
  - Cuando un proceso está ejecutando un procedimiento exportado del monitor decimos que está dentro del monitor.
- **Exclusión mutua**: Si un proceso está ejecutando un procedimiento del monitor (está dentro del mismo), ningún otro proceso podrá entrar al monitor (a cualquier procedimiento del mismo) hasta que el proceso que está dentro del monitor termine la ejecución del procedimiento que ha invocado.
  - Esta política de acceso asegura que las variables permanentes nunca son accedidas concurrentemente.
  - El acceso exclusivo entre los procedimientos del monitor debe estar garantizado en la implementación de los monitores

Funcionamiento de los monitores (2)

- **Los monitores son objetos pasivos**:

  Después de ejecutar el código de inicialización, un monitor es un objeto pasivo y el código de sus procedimientos sólo se ejecuta cuando estos son invocados por los procesos.
- **Instanciación de monitores**: Para cada recurso compartido se tendría que definir un monitor. Para recursos con estructura y comportamiento idénticos podemos instanciar (de forma parametrizada) el monitor tantas veces como recursos se quieran controlar.
  - Declaración de un tipo monitor e instanciación:

```
class monitor nombre_clase_monitor( parametros_formales ) ;
    .... .......... { cuerpo del monitor semejante a los anteriores }
end
var nombre_instancia : nombre_clase_monitor( parametros_actuales ) ;
```

  - **Implementaciones reentrantes**: poder crear una nueva instancia independiente de un monitor para una tarea determinada permite escribir código reentrante que realiza dicha tarea.

Ejemplo de instanciación de un monitor

Podemos escribir un ejemplo similar a `VariableCompartida`, pero ahora parametrizado:

```
{ declaracion del monitor }

class monitor VarComp(pini,pinc : integer);

   var x, inc : integer;

   export incremento, valor;

   procedure incremento( );
   begin
      x := x+inc ;
   end;
   procedure valor(var v : integer);
   begin
      v := x ;
   end;
begin
   x:= pini ; inc := pinc ;
end
```

```
{ ejemplo de uso }

var mv1    : VarComp(0,1);
    mv2    : VarComp(10,4);
    i1,i2 : integer ;
begin
   mv1.incremento() ;
   mv1.valor(i1) ; { i1==1 }
   mv2.incremento() ;
   mv2.valor(i2) ; { i2==14 }
end
```

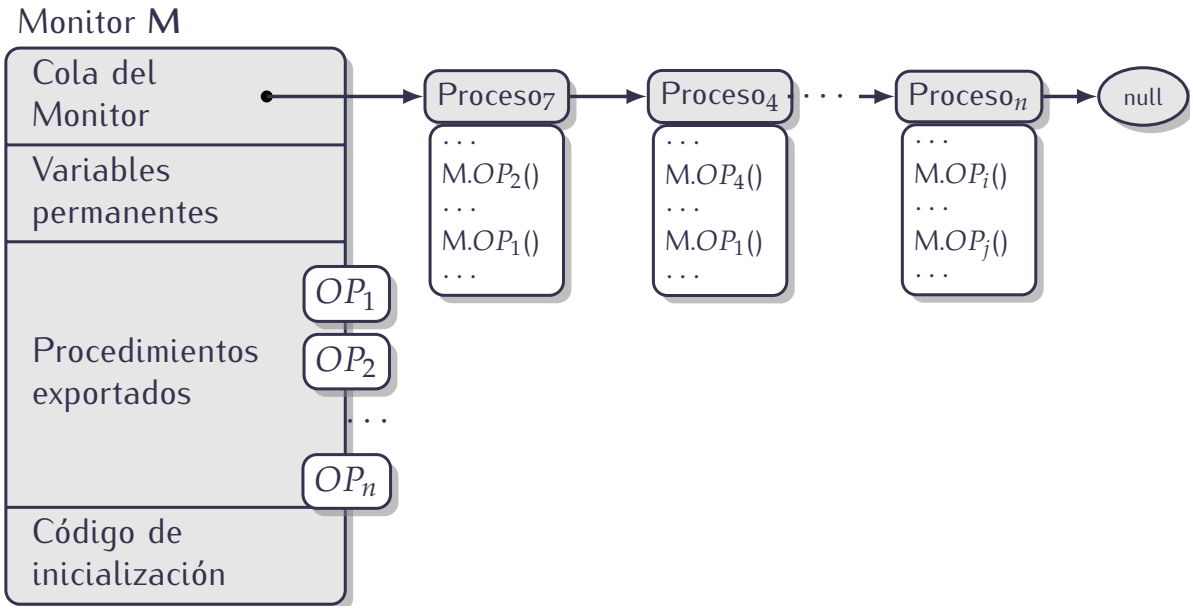
Cola del monitor para exclusión mutua

El control de la exclusión mutua se basa en la existencia de la **cola del monitor**:

- Si un proceso está dentro del monitor y otro proceso intenta ejecutar un procedimiento del monitor, éste último proceso queda bloqueado y se inserta en la cola del monitor
- Cuando un proceso abandona el monitor (finaliza la ejecución del procedimiento), se desbloquea un proceso de la cola, que ya puede entrar al monitor
- Si la cola del monitor está vacía, el monitor está libre y el primer proceso que ejecute una llamada a uno de sus procedimientos, entrará en el monitor
- Para garantizar la vivacidad del sistema, la planificación de la cola del monitor debe seguir una política FIFO

Cola del monitor

Esta cola forma parte del estado del monitor:



3.3.4 Sincronización en monitores

Sincronización en monitores

- Para implementar la sincronización, se requiere de una facilidad para bloqueo-activación de acuerdo a una condición.
- En **semáforos**, las instrucciones de sincronización presentan:
  - Bloqueo-activación
  - Cuenta, para representar la condición.
- En **monitores**:
  - Sólo se dispone de sentencias Bloqueo-activación.
  - La condición se representa mediante los valores de las variables permanentes del monitor.

Primitivas de bloqueo-activación en monitores

- **`wait`**: Estoy esperando a que alguna condición ocurra.
  Bloquea al proceso llamador.
- **`signal`**: Estoy señalando que una condición ocurre.
  Reactiva a un proceso esperando a dicha condición (si no hay ninguno no hace nada).

Ejemplo de monitor para planificar un único recurso

Este monitor permite gestionar el acceso a un recurso que debe usarse en E.M. por parte de varios procesos:

```
monitor recurso;

   var    ocupado   : boolean;
          noocupado : condition;
   export adquirir, liberar ;
```

```
procedure adquirir;
begin
   if ocupado then noocupado.wait();
   ocupado := true;
end;
```

```
procedure liberar;
begin
   ocupado:=false;
   noocupado.signal();
end;
```

```
{ inicializacion }
begin
   ocupado := false;
end
```

Variables condición o señales (1)

Para gestionar las condiciones de sincronización en monitores se utilizan las **señales** o **variables de condición**:

- Debe declararse una variable condición para cualquier condición lógica de sincronización que se pueda dar entre los procesos del programa (por cada razón de bloqueo basada en los valores de las variables permanentes del monitor).
- No tienen valor asociado (no requieren inicialización).
- Representan colas (inicialmente vacías) de procesos esperando la condición.
- Más de un proceso podrá estar dentro del monitor, aunque solo uno estará ejecutándose (el resto estarán bloqueados en variables condición).

Colas de condición o señales (2)

- Dada una variable condición `cond`, se definen al menos las siguientes operaciones asociadas a `cond`:

  Operaciones sobre variables condición

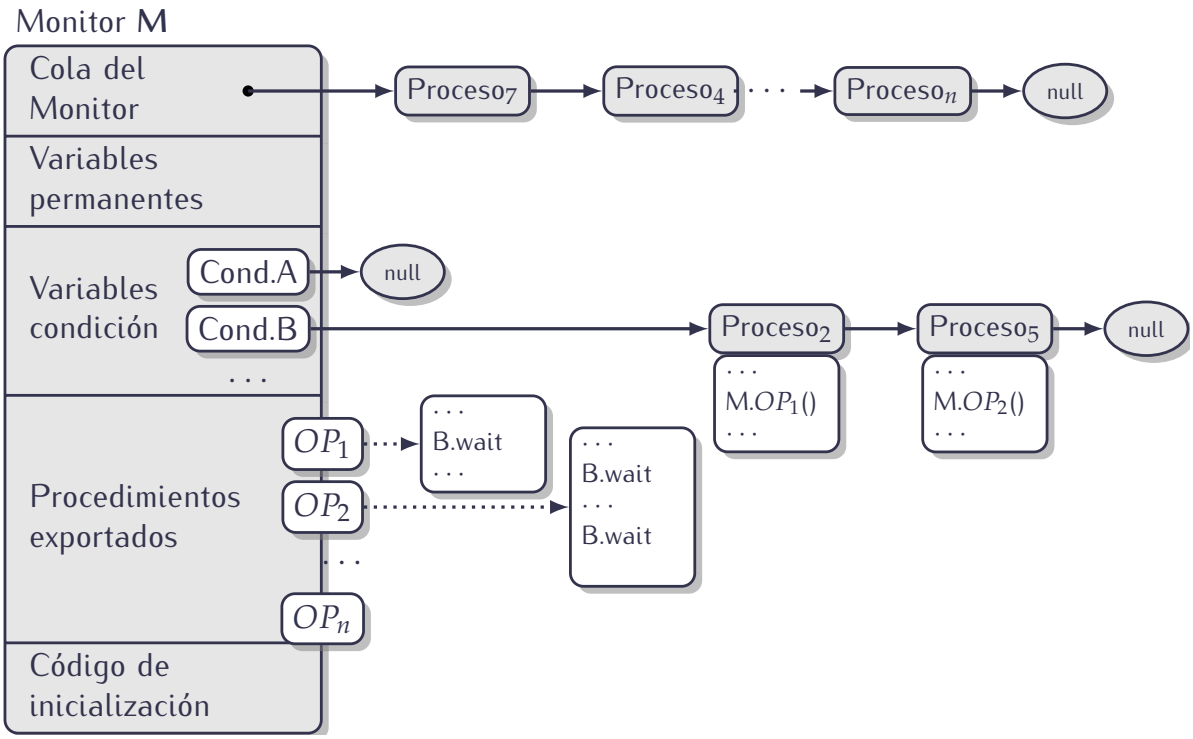
  - `cond.wait()`: Bloquea al proceso que la llama y lo introduce en la cola de la variable condición.
  - `cond.signal()`: Si hay procesos bloqueados por esa condición, libera uno de ellos.
  - `cond.queue()`: Función booleana que devuelve `verdadero` si hay algún proceso esperando en la cola de `cond` y `falso` en caso contrario.

- Cuando hay más de un proceso esperando en una condición `cond`:

- **Propiedad FIFO**: `cond.signal()` reactivará al proceso que lleve más tiempo esperando.
- **Evita la inanición**: Cada proceso en cola obtendrá eventualmente su turno.

Variables condición y colas asociadas.

Los procesos 2 y 5 ejecutan las operaciones 1 y 2, ambas producen esperas de la condición B.



Colas de condición con prioridad

- Por defecto, se usan colas de espera FIFO.
- A veces resulta útil disponer de un mayor control sobre la estrategia de planificación, dando la prioridad del proceso en espera como un nuevo parámetro:

  cond.wait (prioridad:integer)

  - Tras bloqueo, ordena los procesos en cola en base al valor de `prioridad`
  - `cond.signal()` reanuda proceso que especificó el valor más bajo de `prioridad`.

- Precaución al diseñar el monitor: Evitar riesgos como la inanición.
- Ningún efecto sobre la lógica del programa: Funcionamiento similar con/sin colas de prioridad.
- Sólo mejoran las características dependientes del tiempo.

Ejemplo de cola con prioridad(1): Asignador

Asigna un recurso al siguiente trabajo más corto:

```
monitor asignador;

   var    libre : boolean ;
          turno : condition ;
   export peticion, liberar ;
```

```
procedure peticion(tiempo: integer);
begin
   if not libre then
      turno.wait( tiempo );
   libre := false ;
end
```

```
procedure liberar() ;
begin
   libre := true ;
   turno.signal();
end
```

```
{ inicializacion }
begin
   libre := true ;
end
```

Ejemplo de cola con prioridad (2): Reloj con alarma

El proceso llamador se retarda `n` unidades de tiempo:

```
monitor despertador;

   var    ahora      : integer ;
          despertar  : condition ;
   export despiertame, tick ;
```

```
procedure despiertame( n: integer );
   var alarma : integer;
begin
   alarma := ahora + n;
   while ahora<alarma do
      despertar.wait( alarma );
   despertar.signal();
   { por si otro proceso
    coincide en la alarma }
end
```

```
procedure tick();
begin
   ahora := ahora+1 ;
   despertar.signal();
end
```

```
{ Inicializacion }
begin
   ahora := 0 ;
end
```

El problema de los lectores-escritores (LE)

En este ejemplo, hay dos tipos de procesos que acceden concurrentemente a una estructura de datos en memoria compartida:

escritores son procesos que modifican la estructura de datos (escriben en ella). El código de escritura no puede ejecutarse concurrentemente con ninguna otra escritura ni lectura, ya que está formado por una secuencia de instrucciones que temporalmente ponen la estructura de datos en un estado no usable por otros procesos.

lectores son procesos que leen a estructura de datos, pero no modifican su estado en absoluto. El código de lectura puede (y debe) ejecutarse concurrentemente por varios lectores de forma arbitraria, pero no puede hacerse a la vez que la escritura.

la solución de este problema usando semáforos es compleja, veremos que con monitores es sencillo.

LE: vars. permanentes y procedimientos para lectores

```
monitor Lec_Esc ;

   var    n_lec        : integer   ; { numero de lectores leyendo }
          escribiendo  : boolean   ; { true si hay algun escritor escribiendo }
          lectura      : condition ; { no hay escritores escribiendo, lectura posible   }
          escritura    : condition ; { no hay lectores ni escritores, escritura posible }

   export inicio_lectura, fin_lectura,
          inicio_escritura, fin_escritura ;
```

```
procedure inicio_lectura()
begin
   if escribiendo then { si hay escritor }
      lectura.wait() ; { se bloquea }
   n_lec := n_lec + 1 ;
   { desbloqueo en cadena de posibles }
   { procesos lectores bloqueados     }
   lectura.signal()
end
```

```
procedure fin_lectura()
begin
   n_lec:= n_lec - 1;
   if n_lec = 0 then
      { si es el ultimo lector ,    }
      { desbloquear a un escritor }
      escritura.signal()
end
```

LE: procedimientos para escritores

```
procedure inicio_escritura()
begin
 {si hay procs. accediendo: esperar}
 if n_lec > 0 or escribiendo then
   escritura.wait()
 escribiendo:= true;
end;
```

```
procedure fin_escritura()
begin
 escribiendo:= false;
 if lectura.queue() then {si hay lect.:}
  lectura.signal();      {despertar uno }
 else                    {si no hay lect:}
  escritura.signal() ;   {desp. escr.}
end;
```

```
begin { inicializacion }
   n_lec := 0 ;
   escribiendo := false ;
end
```

LE: uso del monitor

Los procesos lectores y escritores tendrían el siguiente aspecto:

```
monitor Lec_Esc ;
   ....
end
```

```
process Lector[ i:1..n ] ;
begin
   while true do begin
     ..........
     Lec_Esc.inicio_lectura() ;
        { codigo de lectura }
     Lec_Esc.fin_lectura() ;
     ..........
   end
end
```

```
process Escritor[ i:1..m ] ;
begin
   while true do begin
      ..........
      Lec_Esc.inicio_escritura() ;
        { codigo de escritura }
      Lec_Esc.fin_escritura() ;
      ..........
   end
end
```

En esta implementación se ha dado prioridad a los lectores (en el momento que un escritor termina, si hay escritores y lectores esperando, pasan los lectores).

3.3.5 Semántica de las señales de los monitores

Efectos de las operaciones sobre la E.M.

- Es necesario liberar la E.M. del monitor justo antes de ejecutar una operación `cond.wait()` para no generar un interbloqueo en la cola de procesos del monitor.
- Cuando se libera un proceso que está esperando por una condición `cond` (proceso señalado) es porque otro proceso (proceso señalador) ha ejecutado una operación `cond.signal()`:
  - El proceso señalado tiene preferencia para acceder al monitor frente a los que esperan en la cola del monitor (ya que éstos podrían cambiar el estado que ha permitido su liberación).

  - Si el proceso señalador continuase "dentro" del monitor, tendríamos una violación de la E.M. del monitor
  - El comportamiento concreto del proceso señalador dependerá de la semántica de señales que se haya establecido en la implementación del monitor.

- Algunos lenguajes implementan una operación que permite liberar a todos los procesos esperando por una condición (**`signal_all`**).

Señalar y continuar (SC)

- El proceso señalador continúa su ejecución dentro del monitor hasta que sale del mismo (porque termina la ejecución del procedimiento) o se bloquea en una condición. En ese momento, el proceso señalado se reactiva y continúa ejecutando código del monitor.
- Al continuar dentro del monitor, el proceso señalador podría cambiar el estado del monitor y hacer falsa la condición lógica por la que esperaba el proceso señalado.
- Por tanto, en el proceso señalado no se puede garantizar que la condición asociada a **`cond`** es cierta al terminar **`cond.wait()`**, y lógicamente es necesario volver a comprobarla entonces.
- Esta semántica obliga a programar la operación **`wait`** en un bucle, de la siguiente manera:

```
while not condicion_lógica_desbloqueo do
   cond.wait() ;
```

Señalar y salir (SS)

- Si hay procesos bloqueados por la condición **`cond`**, el proceso señalador sale del monitor después de ejecutar **`cond.signal()`**.
- En ese caso, la operación **`signal`** conlleva:
  - Liberar al proceso señalado.
  - Terminación del procedimiento del monitor que estaba ejecutando el proceso señalador.
- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó (la condición de desbloqueo se cumple).
- Esta semántica condiciona el estilo de programación ya que obliga a colocar **siempre la operación `signal` como última instrucción** de los procedimientos de monitor que la usen.

Señalar y esperar (SE)

- El proceso señalador se bloquea justo después de ejecutar la operación **`signal`**.
- El proceso señalado entra de forma inmediata en el monitor.

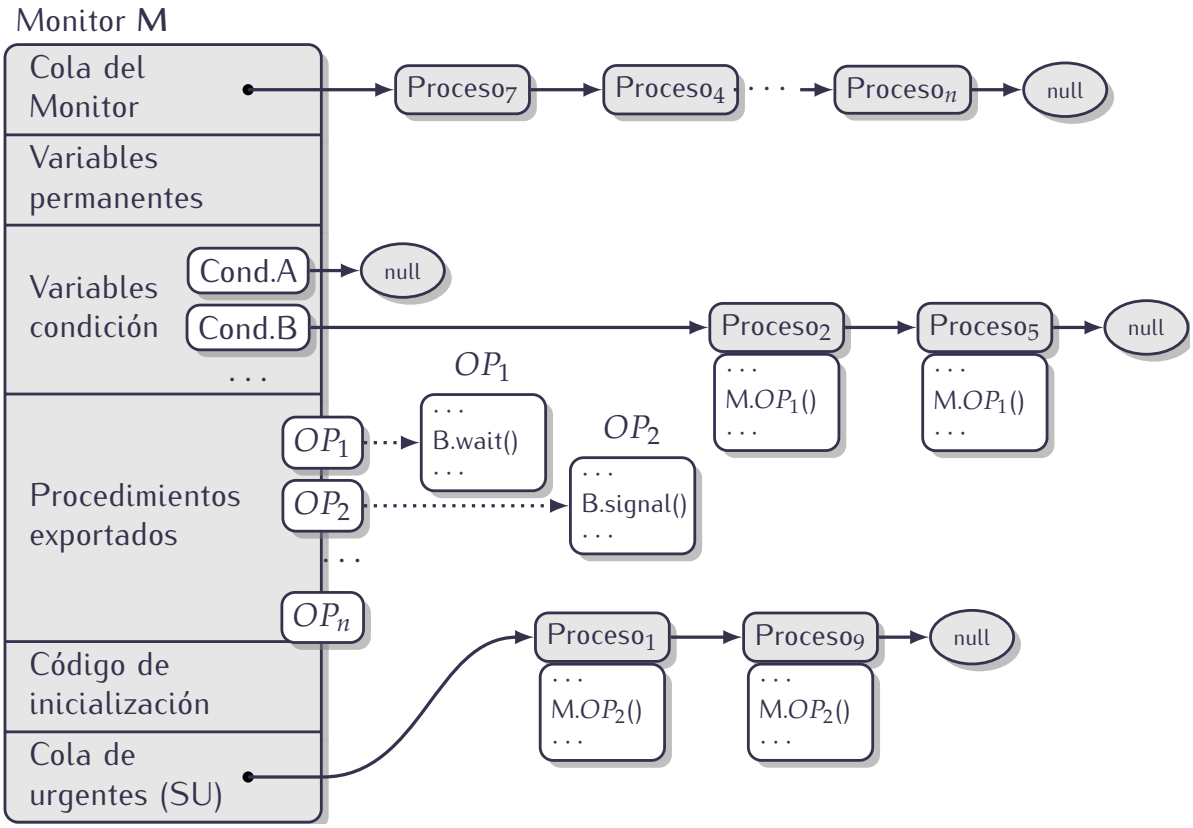
- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó.
- El proceso señalador entra en la cola de procesos del monitor, por lo que está al mismo nivel que el resto de procesos que compiten por la exclusión mutua del monitor.
- Puede considerarse una semántica "injusta" respecto al proceso señalador ya que dicho proceso ya había obtenido el acceso al monitor por lo que debería tener prioridad sobre el resto de procesos que compiten por el monitor.

Señalar y espera urgente (SU)

- Esta solución intenta corregir el problema de falta de equitatividad de la solución anterior.
- El proceso señalador se bloquea justo después de ejecutar la operación **`signal`**.
- El proceso señalado entra de forma inmediata en el monitor.
- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó.
- El proceso señalador entra en una nueva cola de procesos que esperan para acceder al monitor, que podemos llamar cola de procesos urgentes.
- Los procesos de la cola de procesos urgentes tienen preferencia para acceder al monitor frente a los procesos que esperan en la cola del monitor.
- Es la semántica que se supone en los ejemplos vistos.

Procesos en la cola de urgentes.

El proceso 1 y el 9 han ejecutado la op.2, que hace **`signal`** de la cond. B.



Análisis comparativo de las diferentes semánticas

Potencia expresiva

Todas las semánticas son capaces de resolver los mismos problemas.

Facilidad de uso

La semántica **SS** condiciona el estilo de programación y puede llevar a aumentar de forma "artificial" el número de procedimientos.

Eficiencia

- Las semánticas **SE** y **SU** resultan ineficientes cuando la operación signal se ejecuta al final de los procedimientos del monitor.

  El proceso señalador se bloquea dentro del monitor cuando ya no le quedan más instrucciones que ejecutar en el procedimiento, suponiendo un bloqueo innecesario que conlleva un innecesario doble cambio de contexto en CPU (reactivación señalado + salida).

- La semántica **SC** también es un poco ineficiente al obligar a usar un bucle while para cada instrucción **`signal`**.

3.3.6 Implementación de monitores

Implementación de monitores con semáforos

La exclusión mutua en el acceso a los procs. del monitor se puede implementar con un único semáforo **mutex** inicializado a 1:

```
procedure P1(...)
begin
   sem_wait(mutex);
   { cuerpo del procedimiento }
   sem_signal(mutex);
end
```

```
{ inicializacion }
mutex := 1 ;
```

Para implementar la semántica SU necesitamos además un semáforo **next**, para implementar la cola de urgentes, y una variable entera **next_count** para contar los procesos bloqueados en esa cola:

```
procedure P1(...)
begin
   sem_wait(mutex);
   { cuerpo del procedimiento }
   if next_count > 0 then sem_signal(next);
                     else sem_signal(mutex);
end
```

```
{ inicializacion }
next := 0 ;
next_count :=0 ;
```

Implementación de monitores con semáforos

Para implementar las variables condición le asociamos un semáforo a cada una de ellas **x_sem** (inicializado a 0) y una variable para contar los procesos bloqueados en cada una de ellas (**x_sem_count** inicializada a 0):

Implementación de **x.wait()**

```
x_sem_count := x_sem_count + 1 ;
if next_count <> 0 then
   sem_signal(next) ;
else
   sem_signal(mutex);
sem_wait(x_sem);
x_sem_count := x_sem_count - 1 ;
```

Implementación de **x.signal()**

```
if x_sem_count <> 0 then begin
   next_count := next_count + 1 ;
   sem_signal(x_sem);
   sem_wait(next);
   next_count := next_count - 1 ;
end
```

Implementación de monitores con semáforos

Cada monitor tiene asociadas las siguientes colas de procesos:

- **Cola del monitor**: controlada por el semáforo **mutex**.
- **Cola de procesos urgentes**: controlada por el semáforo **next** y el número de procesos en cola se contabiliza en la variable **next_count**. Esta cola solo es necesaria para implementar la semántica SU.

- **Colas de procesos bloqueados en cada condición**: controladas por el semáforo asociado a cada condición (**`x_sem`**) y el número de procesos en cada cola se contabiliza en una variable asociada a cada condición (**`x_sem_count`**).

Esta implementación no permite llamadas recursivas a los procedimientos del monitor y no asegura la suposición FIFO sobre las colas de exclusión mutua y de las señales.

Los semáforos y monitores son equivalentes en potencia expresiva pero los monitores facilitan el desarrollo.

3.3.7 Verificación de monitores

Verificación de programas con monitores

- La verificación de la corrección de un programa concurrente con monitores requiere:
  - Probar la corrección de cada monitor.
  - Probar la corrección de cada proceso de forma aislada.
  - Probar la corrección de la ejecución concurrente de los procesos implicados.
- El programador no puede conocer a priori el orden en que se ejecutarán los procedimientos del monitor.
- El enfoque de verificación que vamos a seguir utiliza un **invariante de monitor**:
  - Establece una relación constante entre los valores permitidos de las variables del monitor.
  - Debe ser cierto cuando un procedimiento empieza a ejecutarse.
  - Debe ser cierto cuando un procedimiento termine de ejecutarse.
  - Debe ser cierto cuando un procedimiento llegue a una llamada a **`wait`**.

Invariante del monitor (IM)

- IM se ha de cumplir después de la inicialización de las variables permanentes.
- IM se ha de cumplir antes y después de cada acción.
- IM se ha de cumplir antes y después de cada operación **`wait`**. Una vez el proceso ha sido liberado (por una operación **`signal`** ejecutada por otro proceso), se debe cumplir la condición que permite liberarlo.
- IM se ha de cumplir antes y después de cada operación **`signal`** sobre una condición. También se debe cumplir antes la condición que permite liberar a un proceso.

3.4 Soluciones software con espera ocupada para E.M.

Introducción

En esta sección veremos diversas soluciones para lograr exclusión mutua en una sección crítica usando variables compartidas entre los procesos o hebras involucrados.

- Estos algoritmos usan dichas variables para hacer espera ocupada cuando sea necesario en el protocolo de entrada.
- Los algoritmos que resuelven este problema no son triviales, y menos para más de dos procesos. En la actualidad se conocen distintas soluciones con distintas propiedades. Veremos estos dos algoritmos:
  - Algoritmo de **Dekker** (para 2 procesos)
  - Algoritmo de **Peterson** (para 2 y para un número arbitrario de procesos).
- Previamente a esos algoritmos, veremos la estructura de los procesos con secciones críticas y las propiedades que deben cumplir los algoritmos.

3.4.1 Estructura de los procesos con Secciones Críticas

Entrada y salida en secciones críticas

Para analizar las soluciones a EM asumimos que un proceso que incluya un bloque considerado como sección crítica (SC) tendrá dicho bloque estructurado en tres etapas:

1. **protocolo de entrada (PE):** una serie de instrucciones que incluyen posiblemente espera, en los casos en los que no se pueda conceder acceso a la sección crítica.
2. **sección crítica (SC):** instrucciones que solo pueden ser ejecutadas por un proceso como mucho.
3. **protocolo de salida (PS):** instrucciones que permiten que otros procesos puedan conocer que este proceso ha terminado la sección crítica.

Todas las sentencias que no forman parte de ninguna de estas tres etapas se denominan **resto de sentencias (RS)** .

Acceso repetitivo a las secciones críticas

En general, un proceso puede contener más de una sección crítica, y cada sección crítica puede estar desglosada en varios bloques de código separados en el texto del proceso. Para simplificar el análisis, suponemos, sin pérdida de generalidad, que:

- Cada proceso tiene una única sección crítica.
- Dicha sección crítica está formada por un único bloque contiguo de instrucciones.
- El proceso es un bucle infinito que ejecuta en cada iteracción dos pasos:
  - Sección crítica (con el PE antes y el PS después)
  - Resto de sentencias: se emplea un tiempo arbitrario no acotado, e incluso el proceso puede finalizar en esta sección, de forma prevista o imprevista.

de esta forma se prevee el caso más general en el cual no se supone nada acerca de cuantas veces un proceso puede intentar entrar en una SC.

Condiciones sobre el comportamiento de los procesos.

Para que se puedan implementar soluciones correctas al problema de EM, es necesario suponer que:

> Los procesos siempre terminan una sección crítica y emplean un intervalo de tiempo finito desde que la comienzan hasta que la terminan.

es decir, durante el tiempo en que un proceso se encuentra en una sección crítica nunca

- Finaliza o aborta.
- Es finalizado o abortado externamente.
- Entra en un bucle infinito.
- Es bloqueado o suspendido indefinidamente de forma externa.

en general, es deseable que el tiempo empleado en las secciones críticas sea el menor posible, y que las instrucciones ejecutadas no puedan fallar.

3.4.2 Propiedades para exclusión mutua

Propiedades requeridas para las soluciones a EM

Para que un algoritmo para EM sea **correcto**, se deben cumplir cada una de estas tres **propiedades mínimas**:

1. **Exclusión mutua**
2. **Progreso**
3. **Espera limitada**

además, hay propiedades deseables adicionales que también deben cumplirse:

4. **Eficiencia**
5. **Equidad**

si bien consideramos correcto un algoritmo que no sea muy eficiente o para el que no pueda demostrarse claramente la equidad.

Exclusión mutua

Es la propiedad fundamental para el problema de la sección crítica. Establece que

> En cada instante de tiempo, y para cada sección crítica existente, habrá como mucho un proceso ejecutando alguna sentencia de dicha región crítica.

En esta sección veremos soluciones de memoria compartida que permiten un único proceso en una sección crítica.

Si bien esta es la propiedad fundamental, no puede conseguirse de cualquier forma, y para ello se establecen las otras dos condiciones mínimas que vemos a continuación.

Progreso

Consideremos una sección crítica en un instante en el cual no hay ningún proceso ejecutándola, pero sí hay uno o varios procesos en el PE compitiendo por entrar a la SC. La propiedad de progreso establece que

> Un algoritmo de EM debe estar diseñado de forma tal que
>
> 1. Después de un intervalo de tiempo finito desde que ingresó el primer proceso al PE, uno de los procesos en el mismo podrá acceder a la SC.
> 2. La selección del proceso anterior es completamente independiente del comportamiento de los procesos que durante todo ese intervalo no han estado en SC ni han intentado acceder.

Cuando la condición (1) no se da, se dice que ocurre un **interbloqueo**, ya que todos los procesos en el PE quedan en espera ocupada indefinidamente sin que ninguno pueda avanzar.

Espera limitada

Supongamos que un proceso emplea un intervalo de tiempo en el PE intentando acceder a una SC. Durante ese intervalo de tiempo, cualquier otro proceso activo puede entrar un número arbitrario de veces n a ese mismo PE y lograr acceso a la SC (incluyendo la posibilidad de que $n = 0$). La propiedad establece que:

> Un algoritmo de exclusión mutua debe estar diseñado de forma que n nunca será superior a un valor máximo determinado.

esto implica que las esperas en el PE siempre serán finitas (suponiendo que los procesos emplean un tiempo finito en la SC).

Propiedades deseables: eficiencia y equidad.

Las propiedades deseables son estas dos:

- **Eficiencia** Los protocolos de entrada y salida deben emplear poco tiempo de procesamiento (excluyendo las esperas ocupadas del PE), y las variables compartidas deben usar poca cantidad de memoria.
- **Equidad** En los casos en que haya varios procesos compitiendo por acceder a una SC (de forma repetida en el tiempo), no debería existir la posibilidad de que sistemáticamente se perjudique a algunos y se beneficie a otros.

3.4.3 Refinamiento sucesivo de Dijkstra

Introducción al refinamiento sucesivo de Dijkstra

El **Refinamiento sucesivo de Dijskstra** hace referencia a una serie de algoritmos que intentan resolver el problema de la exclusión mutua.

- Se comienza desde una versión muy simple, incorrecta (no cumple alguna de las propiedades), y se hacen sucesivas mejoras para intentar cumplir las tres propiedades. Esto ilustra muy bien la importancia de dichas propiedades.

- La versión final correcta se denomina **Algoritmo de Dekker**
- Por simplicidad, veremos algoritmos para 2 procesos únicamente.
- Se asume que hay dos procesos, denominados Proceso 0 y Proceso 1, cada uno de ellos ejecuta un bucle infinito conteniendo:
  1. Protocolo de entrada (PE).
  2. Sección crítica (SC).
  3. Protocolo de salida (PS).
  4. Otras sentencias del proceso (RS).

Versión 1. Pseudocódigo.

En esta versión se usa una variable lógica compartida (**p01sc**) que valdrá **true** solo si el proceso 0 o el proceso 1 están en SC, y valdrá **false** si ninguno lo está:

```
{ variables compartidas y valores iniciales }
var p01sc : boolean := false ; { indica si la SC esta ocupada }
```

```
process P0 ;
begin
   while true do begin
      while p01sc do begin end
      p01sc := true ;
      { seccion critica }
      p01sc := false ;
      { resto seccion }
   end
end
```

```
process P1 ;
begin
   while true do begin
      while p01sc do begin end
      p01sc := true ;
      { seccion critica }
      p01sc := false ;
      { resto seccion }
   end
end
```

Versión 1. Corrección.

Sin embargo, esa versión **no es correcta**. El motivo es que no cumple **la propiedad de exclusión mutua**, pues ambos procesos pueden estar en la sección crítica. Esto puede ocurrir si ambos leen **p01sc** y ambos la ven a **false**. es decir, si ocurre la siguiente secuencia de eventos:

1. el proceso 0 accede al protocolo de entrada, ve **p01sc** con valor **false**,
2. el proceso 1 accede al protocolo de entrada, ve **p01sc** con valor **false**,
3. el proceso 0 pone **p01sc** a **true** y entra en la sección crítica,
4. el proceso 1 pone **p01sc** a **true** y entra en la sección crítica.

Versión 2. Pseudocódigo.

Para solucionar el problema se usará una única variable lógica (**turno0**), cuyo valor servirá para indicar cual de los dos procesos tendrá prioridad para entrar SC la próxima vez que llegen al PE. La variable valdrá **true** si la prioridad es para el proceso 0, y **false** si es para el proceso 1:

```
{ variables compartidas y valores iniciales }
var turno0 : boolean := true ; { podria ser tambien 'false' }
```

```
process P0 ;
begin
   while true do begin
      while not turno0 do begin end
      { seccion critica }
      turno0 := false ;
      { resto seccion }
   end
end
```

```
process P1 ;
begin
   while true do begin
      while turno0 do begin end
      { seccion critica }
      turno0 := true ;
      { resto seccion }
   end
end
```

Versión 2. Corrección.

Esta segunda versión no es tampoco correcta, el motivo es distinto. Se dan estas circunstancias:

- Se cumple la propiedad de exclusión mutua. Esto es fácil de verificar, ya que si un proceso está en SC ha logrado pasar el bucle del protocolo de entrada y por tanto la variable turno0 tiene un valor que forzosamente hace esperar al otro.
- No se cumple la propiedad de **progreso en la ejecución**. El problema está en que este esquema obliga a los procesos a acceder de forma alterna a la sección crítica. En caso de que un proceso quiera acceder dos veces seguidas sin que el otro intente acceder más, la segunda vez quedará esperando indefinidamente.

Este es un buen ejemplo de la necesidad de tener en cuenta cualquier secuencia posible de mezcla de pasos de procesamiento de los procesos a sincronizar.

Versión 3. Pseudocódigo.

Para solucionar el problema de la alternancia, ahora usamos dos variables lógicas (**p0sc**, **p1sc**) en lugar de solo una. Cada variable vale **true** si el correspondiente proceso está en la sección crítica:

```
{ variables compartidas y valores iniciales }
var p0sc  : boolean := false ; { verdadero solo si proc. 0 en SC }
    p1sc  : boolean := false ; { verdadero solo si proc. 1 en SC }
```

```
process P0 ;
begin
   while true do begin
      while p1sc do begin end
      p0sc := true ;
      { seccion critica }
      p0sc := false ;
      { resto seccion }
   end
end
```

```
process P1 ;
begin
   while true do begin
      while p0sc do begin end
      p1sc := true ;
      { seccion critica }
      p1sc := false ;
      { resto seccion }
   end
end
```

Versión 3. Corrección.

De nuevo, esta versión no es correcta:

- Ahora sí se cumple la propiedad de progreso en ejecución, ya que los procesos no tienen que entrar de forma necesariamente alterna, al usar dos variables independientes. Si un proceso quiere entrar, podrá hacerlo si el otro no está en SC.
- No se cumple, sin embargo, la **exclusión mutua**, por motivos parecidos a la primera versión, ya que se pueden producir secuencias de acciones como esta:
  - el proceso 0 accede al protocolo de entrada, ve **p1sc** con valor **falso**,
  - el proceso 1 accede al protocolo de entrada, ve **p0sc** con valor **falso**,
  - el proceso 0 pone **p0sc** a **verdadero** y entra en la sección crítica,
  - el proceso 1 pone **p1sc** a **verdadero** y entra en la sección crítica.

Versión 4. Pseudocódigo.

Para solucionar el problema anterior se puede cambiar el orden de las dos sentencias del PE. Ahora las variables lógicas **p0sc** y **p1sc** están a **`true`** cuando el correspondiente proceso está en SC, pero también cuando está intentando entrar (en el PE):

```
{ variables compartidas y valores iniciales }
var p0sc  : boolean := falso ; { verdadero solo si proc. 0 en PE o SC }
    p1sc  : boolean := falso ; { verdadero solo si proc. 1 en PE o SC }
```

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      while p1sc do begin end
      { seccion critica }
      p0sc := false ;
      { resto seccion }
   end
end
```

```
process P1 ;
begin
   while true do begin
      p1sc := true ;
      while p0sc do begin end
      { seccion critica }
      p1sc := false ;
      { resto seccion }
   end
end
```

Versión 4. Corrección.

De nuevo, esta versión no es correcta:

- Ahora sí se cumple la exclusión mutua. Es fácil ver que si un proceso está en SC, el otro no puede estarlo.
- También se permite el entrelazamiento con regiones no críticas, ya que si un proceso accede al PE cuando el otro no está en el PE ni en el SC, el primero logrará entrar a la SC.
- Sin embargo, no se cumple el **progreso en la ejecución**, ya que puede ocurrir **interbloqueo**. En este caso en concreto, eso puede ocurrir si se produce una secuencia de acciones como esta:
  - el proceso 0 accede al protocolo de entrada, pone **p0sc** a **verdadero**,
  - el proceso 1 accede al protocolo de entrada, pone **p1sc** a **verdadero**,
  - el proceso 0 ve **p1sc** a **verdadero**, y entra en el bucle de espera,
  - el proceso 1 ve **p0sc** a **verdadero**, y entra en el bucle de espera.

Versión 5. Pseudocódigo.

Para solucionarlo, si un proceso ve que el otro quiere entrar, el primero pone su variable temporalmente a **false**:

```
var p0sc : boolean := false ; { true solo si proc. 0 en PE o SC }
    p1sc : boolean := false ; { true solo si proc. 1 en PE o SC }
```

```
1  process P0 ;
2  begin
3     while true do begin
4        p0sc := true ;
5        while p1sc do begin
6           p0sc := false ;
7           { espera durante un tiempo }
8           p0sc := true ;
9        end
10       { seccion critica }
11       p0sc := false ;
12       { resto seccion }
13    end
14 end
```

```
1  process P1 ;
2  begin
3     while true do begin
4        p1sc := true ;
5        while p0sc do begin
6           p1sc := false ;
7           { espera durante un tiempo }
8           p1sc := true ;
9        end
10       { seccion critica }
11       p1sc := false ;
12       { resto seccion }
13    end
14 end
```

Versión 5. Corrección.

En este caso, se cumple exclusión mutua pero, sin embargo, no es posible afirmar que es imposible que se produzca interbloqueo en el PE. Por tanto, no se cumple la propiedad de **progreso**.

- La posibilidad de interbloqueo es pequeña, y depende de cómo se seleccionen las duraciones de los tiempos de la *espera de cortesía*, de cómo se implemente dicha espera, y de la metodología usada para asignar la CPU a los procesos o hebras a lo largo del tiempo.
- En particular, y a modo de ejemplo, el interbloqueo podría ocurrir si ocurre que:
  - Ambos procesos comparten una CPU y la espera de cortesía se implementa como una espera ocupada.
  - Los dos procesos acceden al bucle de las líneas 4-8 (ambas variables están a verdadero).
  - Sistemáticamente, cuando un proceso está en la CPU y ha terminado de ejecutar la asignación de la línea 7, la CPU se le asigna al otro.

3.4.4 Algoritmo de Dekker

Algoritmo de Dekker. Descripción.

El algoritmo de Dekker debe su nombre a su inventor, es un algoritmo correcto (es decir, cumple las propiedades mínimas establecidas), y se puede interpretar como el resultado final del refinamiento sucesivo de Dijkstra:

- Al igual que en la versión 5, cada proceso incorpora una *espera de cortesía* durante la cual le cede al otro la posibilidad de entrar en SC, cuando ambos coinciden en el PE.
- Para evitar interbloqueos, la espera de cortesía solo la realiza uno de los dos procesos, de forma alterna, mediante una variable de turno (parecido a la versión 2).

- La variable de turno permite también saber cuando acabar la espera de cortesía, que se implementa mediante un bucle (espera ocupada).

Algoritmo de Dekker. Pseudocódigo.

```
{ variables compartidas y valores iniciales }
var p0sc   : boolean := falso ; { true solo si proc.0 en PE o SC }
    p1sc   : boolean := falso ; { true solo si proc.1 en PE o SC }
    turno0 : boolean := true  ; { true ==> pr.0 no hace espera de cortesia }
```

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      while p1sc do begin
         if not turno0 then begin
            p0sc := false ;
            while not turno0 do
               begin end
            p0sc := true ;
         end
      end
      { seccion critica }
      turno0 := false ;
      p0sc := false ;
      { resto seccion }
   end
end
```

```
process P1 ;
begin
   while true do begin
      p1sc := true ;
      while p0sc do begin
         if turno0 then begin
            p1sc := false ;
            while turno0 do
               begin end
            p1sc := true ;
         end
      end
      { seccion critica }
      turno0 := true ;
      p1sc := false ;
      { resto seccion }
   end
end
```

3.4.5 Algoritmo de Peterson

Algoritmo de Peterson. Descripción.

Este algoritmo (que también debe su nombre a su inventor), es otro algoritmo correcto para EM, que además es más simple que el algoritmo de Dekker.

- Al igual que el algoritmo de Dekker, usa dos variables lógicas que expresan la presencia de cada proceso en el PE o la SC, más una variable de turno que permite romper el interbloqueo en caso de acceso simultáneo al PE.
- La asignación a la variable de turno se hace al inicio del PE en lugar de en el PS, con lo cual, en caso de acceso simultáneo al PE, el segundo proceso en ejecutar la asignación (atómica) al turno da preferencia al otro (el primero en llegar).
- A diferencia del algoritmo de Dekker, el PE no usa dos bucles anidados, sino que unifica ambos en uno solo.

Pseudocódigo para 2 procesos.

El esquema del algoritmo queda como sigue:

```
{ variables compartidas y valores iniciales }
var p0sc    : boolean := falso ; { true solo si proc.0 en PE o SC }
    p1sc    : boolean := falso ; { true solo si proc.1 en PE o SC }
    turno0  : boolean := true  ; { true ==> pr.0 no hace espera de cortesia }
```

```
1  process P0 ;
2  begin
3     while true do begin
4        p0sc := true ;
5        turno0 := false ;
6        while p1sc and not turno0 do
7           begin end
8        { seccion critica }
9        p0sc := false ;
10       { resto seccion }
11    end
12 end
```

```
1  process P1 ;
2  begin
3     while true do begin
4        p1sc := true ;
5        turno0 := true ;
6        while p0sc and turno0 do
7           begin end
8        { seccion critica }
9        p1sc := false ;
10       { resto seccion }
11    end
12 end
```

Demostración de exclusión mutua. (1/2)

Supongamos que en un instante de tiempo t ambos procesos están en SC, entonces:

(a) La última asignación (atómica) a la variable **turno0** (línea 5), previa a t, finalizó en un instante s (se cumple $s < t$).

(b) En el intervalo de tiempo $(s,t]$, ninguna variable compartida ha podido cambiar de valor, ya que en ese intervalo ambos procesos están en espera ocupada o en la sección crítica y no escriben esas variables.

(c) Durante el intervalo $(s,t]$, las variables **p0sc** y **p1sc** valen **verdadero**, ya que cada proceso puso la suya a **verdadero** antes de s, sin poder cambiarla durante $(s,t]$.

Demostración de exclusión mutua. (2/2)

de las premisas anteriores de deduce que:

(d) Si el proceso 0 ejecutó el último la línea 5 en el instante s, entonces no habría podido entrar en SC entre s y t (la condición de espera del proc.0 se cumpliría en el intervalo $(s,t]$), y por tanto en s forzosamente fue el proceso 1 el último que asignó valor a **turno0**, luego **turno0** vale **verdadero** durante el intervalo $(s,t]$.

(e) la condición anterior implica que el proceso 1 no estaba en SC en s, ni ha podido entrar a SC durante $(s,t]$ (ya que **p0sc y turno0** vale **verdadero**), luego el proceso 1 no está en SC en t.

Vemos que se ha llegado a una contradicción con la hipótesis de partida, que por tanto debe ser falsa, luego no puede existir ningún instante en el cual los dos procesos estén en SC, es decir, se cumple la exclusión mútua.

Espera limitada

Supongamos que hay un proceso (p.ej. el 0) en espera ocupada en el PE, en un instante t, y veamos cuantas veces m puede entrar a SC el proceso 1 antes de que el 0 logre hacerlo:

- El proceso 0 puede pasar a la SC antes que el 1, en ese caso $m = 0$.
- El proceso 1 puede pasar a la SC antes que el 0 (que continúa en el bucle). En ese caso $m = 1$.

En cualquiera de los dos casos, el proceso 1 no puede después llegar (o volver) a SC mientras el 0 continúa en el bucle, ya que para eso el proceso 1 debería pasar antes por la asignación de **verdadero** a **turno0**, y eso provocaría que (después de un tiempo finito) forzosamente el proceso 0 entra en SC mientras el 1 continua en su bucle.
Por tanto, la cota que requiere la propiedad es $n = 1$.

Progreso en la ejecución

Para asegurar el progreso es necesario asegurar

- Ausencia de interbloqueos en el PE: Esto es fácil de demostrar pues si suponemos que hay interbloqueo de los dos procesos, eso significa que son indefinida y simultáneamente verdaderas las dos condiciones de los bucles de espera, y eso implica que es verdad **turno0 y no turno0**, lo cual es absurdo.
- Independencia de procesos en RS: Si un proceso (p.ej. el 0) está en PE y el otro (el 1) está en RS, entonces **p1sc** vale **falso** y el proceso 0 puede progresar a la SC independientemente del comportamiento del proceso 1 (que podría terminar o bloquearse estando en RS, sin impedir por ello el progreso del proc.0). El mismo razonamiento puede hacerse al revés.

Luego es evidente que el algoritmo cumple la condición de progreso.

3.5 Soluciones hardware con espera ocupada (cerrojos) para E.M.

3.5.1 Introducción

Introducción

Los cerrojos constituyen una solución hardware basada en espera ocupada que puede usarse en procesos concurrentes con memoria compartida para solucionar el problema de la exclusión mutua.

- La espera ocupada constituye un bucle que se ejecuta hasta que ningún otro proceso esté ejecutando instrucciones de la sección crítica
- Existe un valor lógico en una posición de memoria compartida (llamado **cerrojo**) que indica si algún proceso está en la sección crítica o no.
- En el protocolo de salida se actualiza el cerrojo de forma que se refleje que la SC ha quedado libre

Veremos una solución elemental que sin embargo es incorrecta e ilustra la necesidad de instrucciones hardware específicas (u otras soluciones más elaboradas).

Una posible solución elemental

Vemos un esquema para procesos que ejecutan SC y RS repetidamente:

```
{ variables compartidas y valores iniciales }
var sc_ocupada : boolean := false ; { cerrojo: verdadero solo si SC ocupada }

{ procesos }
process P[ i : 1 .. n ];
begin
   while true do begin
      while sc_ocupada do begin end
      sc_ocupada := true ;
      { seccion critica }
      sc_ocupada := false ;
      { resto seccion }
   end
end
```

Errores de la solución simple

La solución anterior no es correcta, ya que no garantiza exclusión mutua al existir secuencias de mezclado de instrucciones que permiten a más de un proceso ejecutar la SC a la vez:

- La situación se da si n procesos acceden al protocolo de entrada y todos ellos leen el valor del cerrojo a **false** (ninguno lo escribe antes de que otro lo lea).
- Todos los procesos registran que la SC está libre, y todos acceden a ella.
- El problema es parecido al que ya vimos de acceso simultáneo a una variable en memoria compartida: la lectura y posterior escritura del cerrojo se hace en varias sentencias distintas que se entremezclan.

una solución es usar instrucciones máquina atómicas (indivisibles) para acceso a la zona de memoria donde se aloja el cerrojo. Veremos una de ellas: **LeerAsignar** (**TestAndSet**).

3.5.2 La instrucción LeerAsignar.

La instrucción LeerAsignar.

Es una instrucción máquina disponible en el repertorio de algunos procesadores.

- admite como argumento la dirección de memoria de la variable lógica que actua como cerrojo.
- se invoca como una función desde LLPP de alto nivel, y ejecuta estas acciones:
  1. lee el valor anterior del cerrojo
  2. pone el cerrojo a **verdadero**
  3. devuelve el valor anterior del cerrojo

- al ser una única instrucción máquina su ejecución no puede entremezclarse con ninguna otra instrucción ejecutada en el mismo procesador
- bloquea el bus del sistema y la memoria donde está el cerrojo de forma que ninguna otra instrucción similar de otro procesador puede entremezclarse con ella.

Protocolos de entrada y salida con LeerAsignar

La forma adecuada de usar **LeerAsignar** es la que se indica en este esquema:

```
{ variables compartidas y valores iniciales }
var sc_ocupada : boolean := false ; { true solo si la SC esta ocupada }

{ procesos }
process P[ i : 1 .. n ];
begin
   while true do begin
      while LeerAsignar( sc_ocupada ) do begin end
      { seccion critica }
      sc_ocupada := false ;
      { resto seccion }
   end
end
```

cuando hay más de un proceso intentando entrar en SC (estando SC libre), solo uno de ellos (el primero en ejecutar **LeerAsignar**) ve el cerrojo (**sc_ocupada**) a **falso**, lo pone a **verdadero** y logra entrar a SC.

3.5.3 Desventajas de los cerrojos.

Desventajas de los cerrojos.

Los cerrojos constituyen una solucion válida para EM que consume poca memoria y es eficiente en tiempo (excluyendo las esperas ocupadas), sin embargo:

- las esperas ocupadas consumen tiempo de CPU que podría dedicarse a otros procesos para hacer trabajo útil
- se puede acceder directamente a los cerrojos y por tanto un programa erróneo o escrito malintencionadamente puede poner un cerrojo en un estado incorrecto, pudiendo dejar a otros procesos indefinidamente en espera ocupada.
- en la forma básica que hemos visto no se cumplen ciertas condiciones de equidad

3.5.4 Uso de los cerrojos.

Uso de los cerrojos.

Las desventajas indicadas hacen que el uso de cerrojos sea restringido, en el sentido que:

- por seguridad, normalmente solo se usan desde componentes software que forman parte del sistema operativo, librerías de hebras, de tiempo real o similares (estas componentes suelen estar bien comprobadas y por tanto libres de errores o código malicioso).
- para evitar la pérdida de eficiencia que supone la espera ocupada, se usan solo en casos en los que la ejecución de la SC conlleva un intervalo de tiempo muy corto (por tanto las esperas ocupadas son muy cortas, y la CPU no se desaprovecha).

Bibliografía del tema 2.

Para más información más detallada, ejercicios y bibliografía adicional, se puede consultar:

2.1. **Introducción** Palma (2003) capítulo 3.

2.2. **Monitores como mecanismo de alto nivel.** Palma (2003) capítulo 6, Andrews (2000) capítulo 5, Ben Ari (2006) capítulo 7

2.3. **Soluciones software con Espera Ocupada para E.M.** Palma (2003) capítulo 3, Ben Ari (2006) capítulo 3

2.4. **Soluciones hardware con Espera Ocupada para E.M.** Palma (2003) capítulo 3, Andrews (2000) capítulo 3

2.5. **Semáforos para sincronización.** Palma (2003) capítulo 4, Andrews (2000) capítulo 4, Ben Ari (2006) capítulo 6

3.6 Problemas del tema 2.

9

¿Podría pensarse que una posible solución al problema de la exclusión mutua, sería el siguiente algoritmo que no necesita compartir una variable `Turno` entre los 2 procesos?

(a) ¿Se satisface la exclusión mutua?

(b) ¿Se satisface la ausencia de interbloqueo?

```
{ variables compartidas y valores iniciales }
var c0, c1 : integer ; { los valores iniciales no son relevantes }
```

```
Process P0 ;
begin
   while true do begin
      c0 := 0 ;
      while c1 = 0 do begin
         c0 := 1 ;
         while  c1 = 0 do begin end
         c0 := 0 ;
      end
      { seccion critica }
      c0 := 1 ;
      { resto sentencias }
   end
end
```

```
process P1 ;
begin
   while true do begin
      c1 := 0 ;
      while c0 = 0 do begin
         c1 := 1 ;
         while c0 = 0 do begin end
         c1 := 0 ;
      end
      { seccion critica }
      c1 := 1 ;
      { resto seccion }
   end
end
```

10

Al siguiente algoritmo se le conoce como solución de Hyman al problema de la exclusión mutua. ¿Es correcta dicha solución?

```
{ variables compartidas y valores iniciales }
var c0    : integer := 1 ;
    c1    : integer := 1 ;
    turno : integer := 1 ;
```

```
process P0 ;
begin
   while true do begin
      c0 := 0 ;
      while turno != 0 do begin
         while c1 = 0 do begin end
         turno := 0 ;
      end
      { seccion critica }
      c0 := 1 ;
      { resto sentencias }
   end
end
```

```
process P1 ;
begin
   while true do begin
      c1 := 0 ;
      while turno != 1 do begin
         while c0 = 0 do begin end
         turno := 1 ;
      end
      { seccion critica }
      c1 := 1 ;
      { resto sentencias }
   end
end
```

11

Se tienen 2 procesos concurrentes que representan 2 máquinas expendedoras de tickets (señalan el turno en que ha de ser atendido el cliente), los números de los tickets se representan por dos variables `n1` y `n2`

que valen inicialmente 0. El proceso con el número de ticket más bajo entra en su sección crítica. En caso de tener 2 números iguales se procesa primero el proceso número 1.

a) Demostrar que se verifica la ausencia de bloqueo y la ausencia de inanción.

b) Demostrar que las asignaciones `n1:=1` y `n2:=1` son ambas necesarias.

```
{ variables compartidas y valores iniciales }
var n1 : integer := 0 ;
    n2 : integer := 0 ;
```

```
process P1 ;
begin
   while true do begin
      n1 := 1 ;                          { 1.0 }
      n1 := n2+1 ;                       { 1.1 }
      while n2 != 0 and                  { 1.2 }
            n2 < n1 do begin end;        { 1.3 }
      { seccion critica }
      n1 := 0 ;                          { 1.4 }
      { resto sentencias }
   end
end
```

```
process P2 ;
begin
   while true do begin
      n2 := 1 ;                          { 2.0 }
      n2 := n1+1 ;                       { 2.1 }
      while n1 != 0  and                 { 2.2 }
            n1 <= n2 do begin end;       { 2.3 }
      { seccion critica }
      n2 := 0 ;                          { 2.4 }
      { resto sentencias }
   end
end
```

12

El siguiente programa es una solución al problema de la exclusión mutua para 2 procesos. Discutir la corrección de esta solución: si es correcta, entonces probarlo. Si no fuese correcta, escribir escenarios que demuestren que la solución es incorrecta.

```
{ variables compartidas y valores iniciales }
var c0 : integer := 1 ;
    c1 : integer := 1 ;
```

```
process P0 ;
begin
   while true do begin
      repeat
         c0 := 1-c1 ;
      until c1 != 0 ;
      { seccion critica }
      c0 := 1 ;
      { resto sentencias }
   end
end
```

```
process P1 ;
begin
   while true do begin
      repeat
         c1 := 1-c0 ;
      until c0 != 0 ;
      { seccion critica }
      c1 := 1 ;
      { resto sentencias }
   end
end
```

13

Diseñar una solución hardware basada en espera ocupada para el problema de la exclusión mutua utilizando la instrucción máquina **`swap(x,y)`** (en lugar de usar **`LeerAsignar`**) cuyo efecto es intercambiar los dos valores lógicos almacenados en las posiciones de memoria `x` e `y`.

14

Supongamos que tres procesos concurrentes acceden a dos variables compartidas (`x` e `y`) según el siguiente esquema:

```
var x, y : integer ;
```

```
{ accede a 'x' }
process P1 ;
begin
 while true do begin
   x := x+1 ;
   { .... }
 end
end
```

```
{ accede a 'x' e 'y' }
process P2 ;
begin
 while true do begin
   x := x+1 ;
   y := x ;
   { .... }
 end
end
```

```
{ accede a 'y' }
process P3 ;
begin
 while true do begin
   y := y+1 ;
   { .... }
 end
end
```

con este programa como referencia, realiza estas dos actividades:

1. usando un único semáforo para exclusión mutua, completa el programa de forma que cada proceso realice todos sus accesos a `x` e `y` sin solaparse con los otros procesos (ten en cuenta que el proceso 2 debe escribir en `y` el mismo valor que acaba de escribir en `x`).

2. la asignación `x:=x+1` que realiza el proceso 2 puede solaparse sin problemas con la asignación `y:=y+1` que realiza el proceso 3, ya que son independientes. Sin embargo, en la solución anterior, al usar un único semáforo, esto no es posible. Escribe una nueva solución que permita el solapamiento descrito, usando dos semáforos para dos secciones críticas distintas (las cuales, en el proceso 2, aparecen anidadas).

15

En algunas aplicaciones es necesario tener exclusión mutua entre procesos con la particularidad de que puede haber como mucho n procesos en una sección crítica, con n arbitrario y fijo, pero no necesariamente igual a la unidad sino posiblemente mayor. Diseña una solución para este problema basada en el uso de espera ocupada y cerrojos. Estructura dicha solución como un par de subrutinas (usando una misma estructura de datos en memoria compartida), una para el protocolo de entrada y otro el de salida, e incluye el pseudocódigo de las mismas.

16

Sean los procesos P_1, P_2 y P_3, cuyas secuencias de instrucciones son las que se muestran en el cuadro. Resuelva los siguientes problemas de sincronización (son independientes unos de otros):

a) P_2 podrá pasar a ejecutar e solo si P_1 ha ejecutado a o P_3 ha ejecutado g.

b) P_2 podrá pasar a ejecutar e solo si P_1 ha ejecutado a y P_3 ha ejecutado g.

c) Solo cuando P_1 haya ejecutado b, podrá pasar P_2 a ejecutar e y P_3 a ejecutar h.

d) Sincroniza los procesos de forma que las secuencias b en P_1, f en P_2, y h en P_3, sean ejecutadas como mucho por dos procesos simultáneamente.

```
{ variables globales }
```

```
process P1 ;
begin
   while true do begin
      a
      b
      c
   end
end
```

```
process P2 ;
begin
   while true do begin
      d
      e
      f
   end
end
```

```
process P3 ;
begin
   while true do begin
      g
      h
      i
   end
end
```

17

El cuadro que sigue nos muestra dos procesos concurrentes, P_1 y P_2, que comparten una variable global x (las restantes variables son locales a los procesos).

a) Sincronizar los procesos para que P_1 use todos los valores x suministrados por P_2.

b) Sincronizar los procesos para que P_1 utilice un valor sí y otro no de la variable x, es decir, utilice los valores primero, tercero, quinto, etc...

```
{ variables globales }
```

```
process P1 ;
   var m : integer ;
begin
   while true do begin
      m := 2*x-n ;
      print( m );
   end
end
```

```
process P2
   var d : integer ;
begin
   while true do begin
      d := leer_teclado();
      x := d-c*5 ;
   end
end
```

18

Aunque un monitor garantiza la exclusión mutua, los procedimientos tienen que ser reentrantes. Explicar porqué.

19

Se consideran dos recursos denominados r_1 y r_2. Del recurso r_1 existen N_1 ejemplares y del recurso r_2 existen N_2 ejemplares. Escribir un monitor que gestione la asignación de los recursos a los procesos de usuario, suponiendo que cada proceso puede pedir:

- Un ejemplar del recurso r_1.
- Un ejemplar del recurso r_2.
- Un ejemplar del recurso r_1 y otro del recurso r_2.

La solución deberá satisfacer estas dos condiciones:

- Un recurso no será asignado a un proceso que demande un ejemplar de r_1 o un ejemplar de r_2 hasta que al menos un ejemplar de dicho recurso quede libre.
- Se dará prioridad a los procesos que demanden un ejemplar de ambos recursos.
- Se asume semántica SU.

20

Escribir una solución al problema de *lectores-escritores* con monitores:

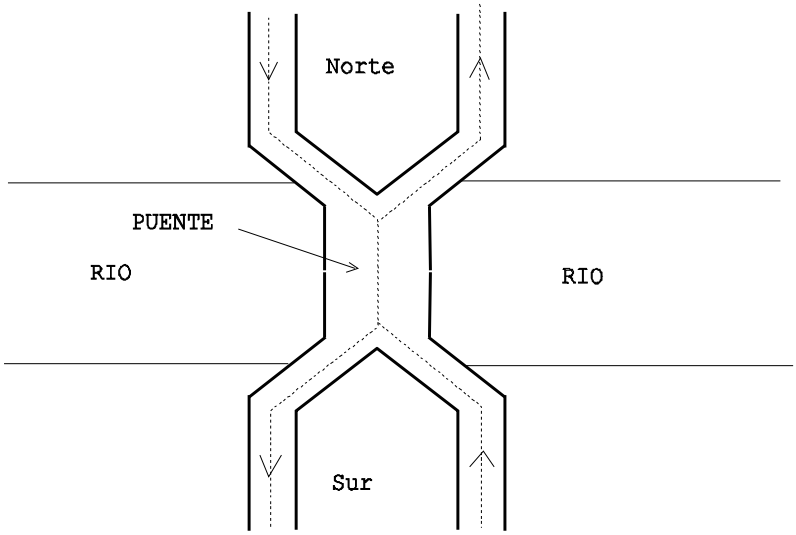
a) Con prioridad a los lectores.

b) Con prioridad a los escritores.

c) Con prioridades iguales.

21

Varios coches que vienen del norte y del sur pretenden cruzar un puente sobre un río. Solo existe un carril sobre dicho puente. Por lo tanto, en un momento dado, el puente solo puede ser cruzado por uno o más coches en la misma dirección (pero no en direcciones opuestas).



a) Completar el código del siguiente monitor que resuelve el problema del acceso al puente suponiendo que llega un coche del norte (sur) y cruza el puente si no hay otro coche del sur (norte) cruzando el puente en ese momento.

```
Monitor Puente

var  ... ;

procedure EntrarCocheDelNorte()
begin
   ...
end
procedure SalirCocheDelNorte()
begin
   ....
end
procedure EntrarCocheDelSur()
begin
   ....
end
procedure SalirCocheDelSur()
begin
  ...
end

{ Inicializacion }
begin
   ....
end
```

b) Mejorar el monitor anterior, de forma que la dirección del trafico a través del puente cambie cada vez que lo hayan cruzado 10 coches en una dirección, mientras 1 ó más coches estuviesen esperando cruzar el puente en dirección opuesta.

22

Una tribu de antropófagos comparte una olla en la que caben M misioneros. Cuando algún salvaje quiere comer, se sirve directamente de la olla, a no ser que ésta esté vacía. Si la olla está vacía, el salvaje despertará al cocinero y esperará a que éste haya rellenado la olla con otros M misioneros.

Para solucionar la sincronizacion usamos un monitor llamado **Olla**, que se puede usar así:

```
monitor Olla ;
....
begin
   ....
end
```

```
process ProcSalvaje[ i:1..N ] ;
begin
   while true do begin
      Olla.Servirse_1_misionero();
      Comer(); { es un retraso aleatorio }
   end
end
```

```
process ProcCocinero ;
begin
   while true do begin
      Olla.Dormir();
      Olla.Rellenar_Olla();
   end
end
```

Diseña el código del monitor **Olla** para la sincronización requerida, teniendo en cuenta que:

- La solución no debe producir interbloqueo.
- Los salvajes podrán comer siempre que haya comida en la olla,
- Solamente se despertará al cocinero cuando la olla esté vacía.

23

Una cuenta de ahorros es compartida por varias personas (procesos). Cada persona puede depositar o retirar fondos de la cuenta. El saldo actual de la cuenta es la suma de todos los depósitos menos la suma de todos los reintegros. El saldo nunca puede ser negativo.

Queremos usar un monitor para resolver el problema. El monitor debe tener 2 procedimientos: **depositar**(c) y **retirar**(c). Suponer que los argumentos de las 2 operaciones son siempre positivos, e indican las cantidades a depositar o retirar. Hacerlo de estas dos formas:

(a) Todo proceso puede retirar fondos mientras la cantidad solicitada c sea menor o igual que el saldo disponible en la cuenta en ese momento. Si un proceso intenta retirar una cantidad c mayor que el saldo, debe quedar bloqueado hasta que el saldo se incremente lo suficiente (como consecuencia de que otros procesos depositen fondos en la cuenta) para que se pueda atender la petición. Hacer dos versiones:

(a.1) colas normales (FIFO), sin prioridad.

(a.2) con colas de prioridad.

(b) El reintegro de fondos a los clientes se hace únicamente según el orden de llegada, por lo cual si un cliente retira entnces dicho cliente es el que más tiempo lleva esperando en el banco. Por ejemplo, suponer que el saldo es 200 unidades y un cliente está esperando un reintegro de 300 unidades. Si llega otro cliente debe esperarse, incluso si quiere retirar 200 unidades. De nuevo, resolverlo de dos formas:

(b.1) colas normales (FIFO), sin prioridad.

(b.2) con colas de prioridad.

24

Los procesos P_1, P_2,...,P_n comparten un único recurso R, pero solo un proceso puede utilizarlo cada vez. Un proceso P_i puede comenzar a utilizar R si está libre; en caso contrario, el proceso debe esperar a que el recurso sea liberado por otro proceso. Si hay varios procesos esperando a que quede libre R, se concederá al proceso que tenga mayor prioridad. La regla de prioridad de los procesos es la siguiente: el proceso P_i tiene prioridad i, (con $1 \leq i \leq n$), donde los números menores implican mayor prioridad. Implementar un monitor que implemente los procedimientos **`Pedir`** y **`Liberar`**.

25

En un sistema hay dos tipos de procesos: A y B. Queremos implementar un esquema de sincronización en el que los procesos se sincronizan por bloques de 1 proceso del tipo A y 10 procesos del tipo B. De acuerdo con este esquema:

- Si un proceso de tipo A llama a la operación de sincronización, y no hay (al menos) 10 procesos de tipo B bloqueados en la operación de sincronización, entonces el proceso de tipo A se bloquea.
- Si un proceso de tipo B llama a la operación de sincronización, y no hay (al menos) 1 proceso del tipo A y 9 procesos del tipo B (aparte de él mismo) bloqueados en la operación de sincronización, entonces el proceso de tipo B se bloquea.
- Si un proceso de tipo A llama a la operación de sincronización y hay (al menos) 10 procesos bloqueados en dicha operación, entonces el proceso de tipo A no se bloquea y además deberán desbloquearse exactamente 10 procesos de tipo B. Si un proceso de tipo B llama a la operación de sincronización y hay (al menos) 1 proceso de tipo A y 9 procesos de tipo B bloqueados en dicha operación, entonces el proceso de tipo B no se bloquea y además deberán desbloquearse exactamente 1 proceso del tipo A y 9 procesos del tipo B.
- No se requiere que los procesos se desbloqueen en orden FIFO.

los procedimientos para `pedir` y `liberar` el recurso

Implementar un Monitor que implemente procedimientos para llevar a cabo la sincronización requerida entre los diferentes tipos de procesos (el monitor puede exportar una única operación de sincronización para todos los tipos de procesos o una operación espcífica para los de tipo A y otra para los de tipo B.

26

El siguiente monitor (**Barrera2**) proporciona un único procedimiento de nombre **entrada** que provoca que el primer proceso que lo llama sea suspendido y el segundo que lo llama despierte al primero que lo llamó, y así actúa cíclicamente. Obtener una implementación de este monitor usando semáforos.

```
Monitor Barrera2 ;

   var n : integer;       { num. de proc. que han llegado desde el signal }
       s : condicion ;    { cola donde espera el segundo                  }

procedure entrada() ;
begin
  n := n+1 ;          { ha llegado un proceso mas }
  if n<2 then         { si es el primero:         }
     s.wait()         {   esperar al segundo      }
  else begin          { si es el segundo:         }
     n := 0;          {   inicializa el contador  }
     s.signal()       {   despertar al primero    }
  end
end
{  Inicializacion  }
begin
   n := 0 ;
end
```

27

Problema de los filósofos-comensales. Sentados a una mesa están cinco filósofos. La actividad de cada filósofo es un ciclo sin fin de las operaciones de pensar y comer. Entre cada dos filósofos hay un tenedor. Para comer, un filósofo necesita obligatoriamente dos tenedores: el de su derecha y el de su izquierda. Se han definido cinco procesos concurrentes, cada uno de ellos describe la actividad de un filósofo. Los procesos usan un monitor, llamado **MonFilo**.

Antes de comer cada filósofo debe disponer de su tenedor de la derecha y el de la izquierda, y cuando termina la actividad de comer, libera ambos tenedores. El filósofo i alude al tenedor de su derecha como el número i, y al de su izquierda como el número $i+1 \mod 5$.

El monitor **MonFilo** exportará dos procedimentos: **coge_tenedor**(num_tenedor,num_proceso) y **libera_tenedor**(n
para indicar que un proceso filósofo desea coger un tenedor determinado.

El código del programa es el siguiente:

```
monitor MonFilo ;
   ....
begin
```

```
   ....
end

process Filosofo[ i: 0..4 ] ;
begin
   while true do begin
      MonFilo.coge_tenedor(i,i);           { argumento 1=codigo tenedor    }
      MonFilo.coge_tenedor(i+1 mod 5,i);   { argumento 2=numero de proceso }
      comer();
      MonFilo.libera_tenedor(i);
      MonFilo.libera_tenedor(i+1 mod 5);
      pensar();
   end
end
```

Diseña el monitor **MonFilo**, compartido por todos los procesos filósofos. Para evitar interbloqueo (que ocurre si todos los filósofos toman el tenedor a su izquierda antes de que ninguno de ellos pueda coger el tenedor de la derecha), la solución no debe permitir que haya más de cuatro filósofos simultáneamente intentado coger un tenedor

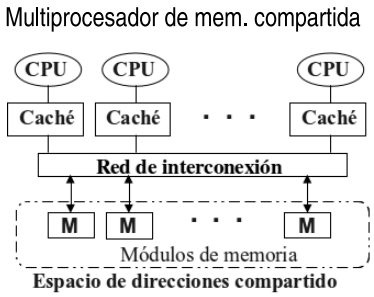
Chapter 4

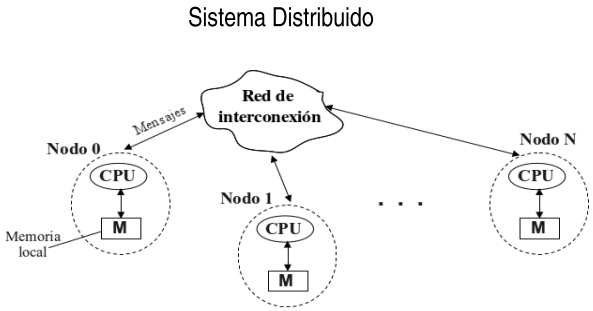
Tema 3. Sistemas basados en paso de mensajes.

4.1 Mecanismos básicos en sistemas basados en paso de mensajes

4.1.1 Introducción

Introducción. Multiprocesador con mem. compartida vs. Sistema Distribuido





- **Multiprocesador**:
  - Más fácil programación (variables compartidas): se usan mecanismos como cerrojos, semáforos y monitores.
  - Implementación más costosa y escalabilidad hardware limitada: el acceso a memoria común supone un cuello de botella.

- **Sistema Distribuido**:
  - Distribución de datos y recursos.
  - Soluciona el problema de la escalabilidad y el elevado coste.
  - Mayor dificultad de programación: No hay direcciones de memoria comunes y mecanismos como los monitores son inviables.

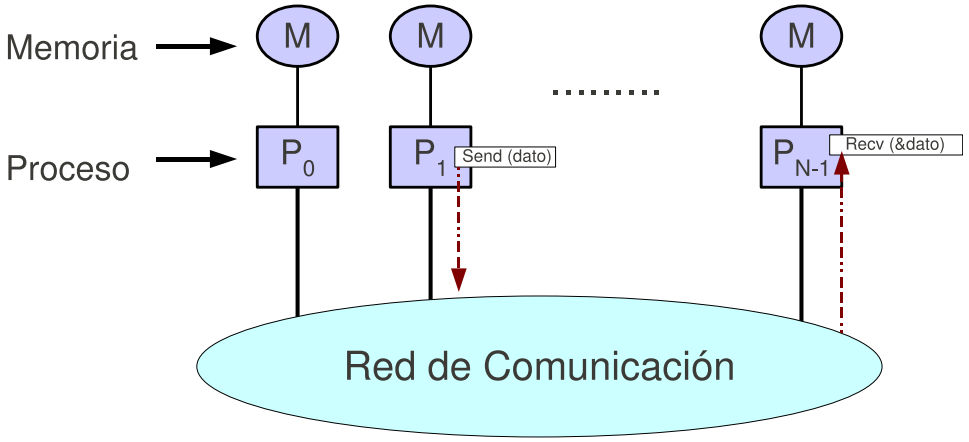
Necesidad de una notación de progr. distribuida

- Lenguajes tradicionales (memoria común)
  - **Asignación**: Cambio del estado interno de la máquina.
  - **Estructuración**: Secuencia, repetición, alternación, procedimientos, etc.
- Extra añadido: **Envío / Recepción** $\Longrightarrow$ Afectan al entorno externo.
  - Son tan importantes como la asignación.
  - Permiten comunicar procesos que se ejecutan en paralelo.
- **Paso de mensajes**:
  - **Abstracción**: Oculta Hardware (red de interconexión).
  - **Portabilidad**: Se puede implementar eficientemente en cualquier arquitectura (multiprocesador o plataforma distribuida).
  - No requiere mecanismos para asegurar la exclusión mutua.

4.1.2 Vista logica arquitectura y modelo de ejecución

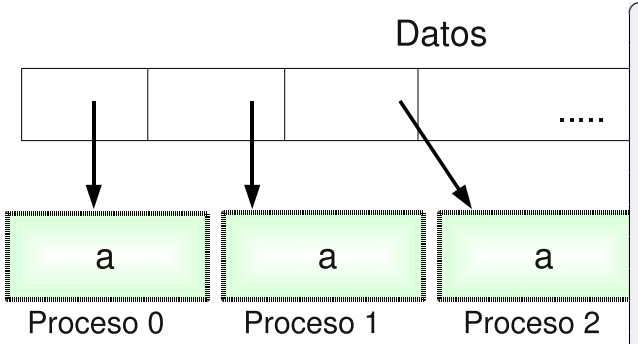
Vista logica de la arquitectura

- N procesos, cada uno con su espacio de direcciones propio (memoria).
- En un mismo procesador podrían residir físicamente varios procesos, aunque es muy común la ejecución 1 proceso-procesador.
- Los procesos se comunican mediante envío y recepción de mensajes.
- **Cada interacción requiere cooperación entre 2 procesos**: el propietario de los datos (emisor) debe intervenir aunque no haya conexión lógica con el evento tratado en el receptor.



Estructura de un programa de paso de mensajes. SPMD

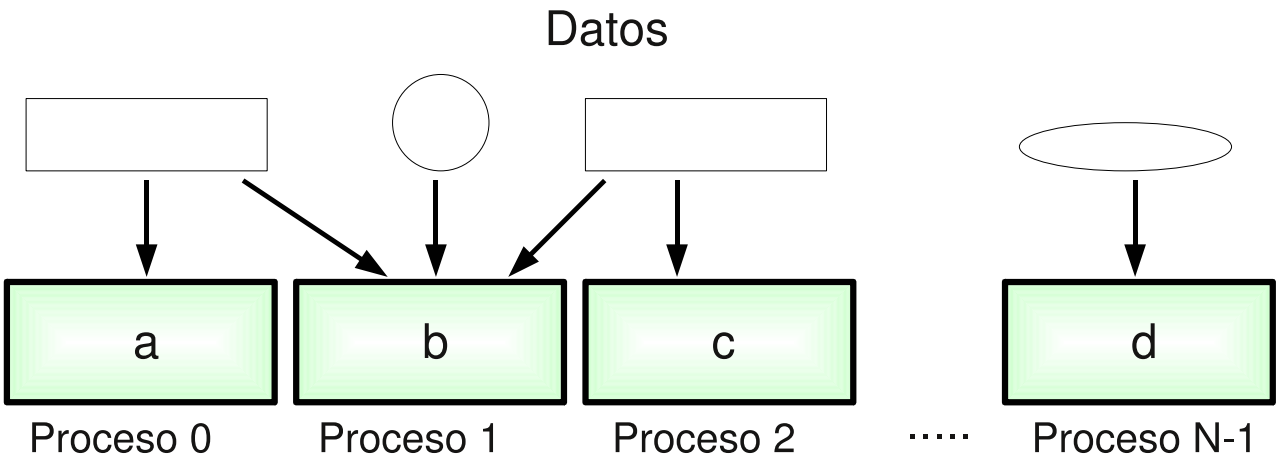
- Ejecutar un programa diferente sobre cada proceso puede ser algo difícil de manejar.
- **Enfoque común**: **Estilo SPMD (Single Program Multiple Data)**.
  - El código que ejecutan los diferentes procesos es idéntico, pero sobre datos diferentes.
  - El código incluye la lógica interna para que cada tarea ejecute lo que proceda dependiendo de la identidad del proceso.




```
{ Proceso[0] envia y Proceso[1] recibe }
{ el codigo es el mismo }
process Proceso[ n_proc : 0..1 ];
   var dato : integer ;
begin
   if n_proc == 0 then begin {si soy 0}
      dato := Produce();
      send( dato, Proceso[1]);
   end else begin {si soy 1}
      receive( dato, Proceso[0] );
      Consume( dato );
   end
end
```

Estructura de un programa de paso de mensajes. MPMD

- **Estilo MPMD (Multiple Program Multiple Data)**:
  - Cada proceso ejecuta el mismo o diferentes programas de un conjunto de ficheros objeto ejecutables.
  - Los diferentes procesos pueden usar datos diferentes.



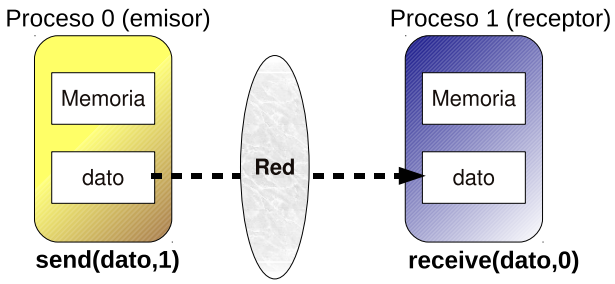
```
process ProcesoA ;
   var dato_env : integer ;
begin
   dato_env := Produce();
   send( dato_env, ProcesoB );
end
```

```
process ProcesoB ;
   var dato_rec : integer ;
begin
   receive( dato_rec, ProcesoA );
   Consume( dato_rec );
end
```

4.1.3 Primitivas básicas de paso de mensajes

Primitivas básicas de paso de mensajes

- **Envío:** **`send`**(*expresion*, *identificador_proceso_destino*)
- **Recepción:** **`receive`**(*variable*, *identificador_proceso_origen*)



Las primitivas **`send`** y **`receive`** permiten:

- **Comunicación**: Los procesos envían/reciben mensajes en lugar de escribir/leer en variables compartidas.
- **Sincronización**: la recepción de un mensaje es posterior al envío. Generalmente, la recepción supone la espera del mensaje por parte del receptor.

Esquemas de identificación de la comunicación

¿Cómo identifica el emisor al receptor del mensaje y viceversa?
Existen dos posibilidades:

Denominación directa

- Emisor identifica explícitamente al receptor y viceversa.
- Se utilizan los identificadores de los procesos.

Denominación indirecta

Los mensajes se depositan en almacenes intermedios que son globales a los procesos (buzones).

Denominación directa

- **Ventaja principal**: No hay retardo para establecer la identificación.
- **Inconvenientes**:
  - Cambios en la identificación requieren recompilar el código.
  - Sólo permite enlaces 1-1.

```
process P0 ;
var dato : integer ;
begin
   dato := Produce();
   send( dato, P1 );
end
```

```
process P1 ;
var midato : integer ;
begin
   receive( midato, P0 );
   Consume( midato );
end
```

- Existen **esquemas asimétricos**: el emisor identifica al receptor, pero el receptor no indica emisor:
  - **receive**(*variable*, *id_origen*): en *id_origen* se devuelve el emisor del mensaje.
  - **receive**(*variable*, *ANY*): *ANY* indica que se puede recibir de cualquier emisor.

Denominación indirecta

- Presenta una mayor **flexibilidad** al permitir la comunicación simultánea entre varios procesos.
- **Declaración estática/dinámica**:
  - Estática: fija fuente/destino en tiempo de compilación. No apropiada en entornos cambiantes.
  - Dinámica: más potente pero menos eficiente

```
var buzon : channel of integer ;
```

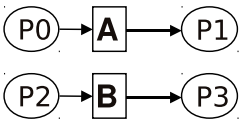
```
process P0 ;
var dato : integer ;
begin
   dato := Produce();
   send( dato, buzon );
end
```

```
process P1 ;
var midato : integer ;
begin
   receive( midato, buzon );
   Consume( midato );
end
```

Denominación indirecta (2)

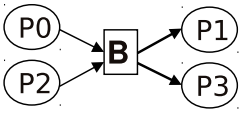
Existen tres tipos de buzones: canales (uno a uno), puertos (muchos a uno) y buzones generales (muchos a muchos):

Canales (1 a 1)



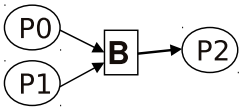
- Tipo fijo

Buzones generales (n a n)



- Destino: *send* de cualquier proc.
- Fuente: *receive* de cualquier proc.
- Implementación complicada.
  - •Enviar mensaje y transmitir todos los lugares.
  - •Recibir mensaje y notificar recepción a todos.

Puertos (n a 1)



- Destino: Un único proceso
- Fuente: Varios procesos
- Implementación más sencilla.

- Un mensaje enviado a un buzón general permanece en dicho buzón hasta que haya sido leído por todos los procesos receptores potenciales (cada envío es un *broadcast*).

Comportamiento de las operaciones de paso de mensajes

- El comportamiento de las operaciones puede variar dependiendo de las necesidades. Ejemplo:

```
process Emisor ;
   var a : integer := 100 ;
begin
   send( a, Receptor ) ;
   a := 0 ;
end
```

```
process Receptor ;
   var b : integer := 1 ;
begin
   receive( b, Emisor ) ;
   imprime( b );
end
```

- El comportamiento esperado del **send** impone que el valor recibido en `b` sea el que tenía `a` (100) justo antes de la llamada a **send**.
- Si se garantiza esto, se habla de comportamiento *seguro* (se habla de que el programa de paso de mensajes es seguro).
- Generalmente, un programa no seguro (en el ejemplo, si pudiera imprimirse 0 o 1 en lugar de 100), se considera incorrecto, aunque existen situaciones en las que puede interesar usar operaciones que no garantizan dicha seguridad.

Instantes críticos en las operaciones de paso de mensajes

```
process Emisor ;
   var a : integer := 100 ;
begin
   send( a, Receptor ) ;
end
```

```
process Receptor ;
   var b : integer ;
begin
   receive( b, Emisor ) ;
   imprime( b );
end
```

Instantes relevantes en el emisor:

- E_1 : el emisor acaba de indicar al Sistema de Paso de Mensajes (SPM) la direccion de `a` y el identificador del receptor.
- E_2 : el SPM comienza la lectura de datos en `a`.
- E_3 : el SPM termina de leer todos los datos en `a`.

Instantes relevantes en el receptor:

- R_1 : el receptor acaba de indicar al SPM la direccion de `b` y el identificador del emisor.
- R_2 : el SPM comienza a escribir en `b`.
- R_3 : el SPM termina de escribir en `b`.

Ordenación de los instantes relevantes

Los eventos relevantes citados deben ocurrir en cierto orden:

- El emisor puede comenzar el envio antes que el receptor, o al revés. Es decir, no hay un orden temporal entre E_1 y R_1 (se cumple $E_1 <= R_1$ o $R_1 <= E_1$).
- Lógicamente, los tres eventos del emisor se producen en orden, y también los tres del receptor. Se cumple que: $\underline{E_1 < E_2 < E_3}$ y también que $\underline{R_1 < R_2 < R_3}$.
- Antes de que se escriba el primer byte en el receptor, se debe haber comenzado ya la lectura en el emisor, por tanto $\underline{E_2 < R_2}$.
- Antes de que se acaben de escribir los datos en el receptor, se deben haber acabado de leer en el emisor, es decir $\underline{E_3 < R_3}$.
- Por transitividad se puede demostrar que R_3 **es siempre el último evento**.

Mensajes en tránsito

Por la hipótesis de progreso finito, el intervalo de tiempo entre E_1 y R_3 tiene una duración no predecible. Entre E_1 y R_3, se dice que el **mensaje está en tránsito**.

- El SPM necesita usar memoria temporal para todos los mensajes en tránsito que esté gestionando en un momento dado.

- La cantidad de memoria necesaria dependerá de diversos detalles (tamaño y número de los mensajes en tránsito, velocidad de la transmisión de datos, políticas de envio de mensajes, etc...)
- Dicha memoria puede estar ubicada en el nodo emisor y/o en el receptor y/o en nodos intermedios, si los hay.
- En un momento dado, el SPM puede detectar que no tiene suficiente memoria para almacenamiento temporal de datos durante el envio (debe retrasar la lectura en el emisor hasta asegurarse que hay memoria para enviar los datos)

Seguridad de las operaciones de paso de mensajes

Las operaciones de envío y/o recepción podrían **no ser seguras**, es decir, el valor que el emisor pretendía enviar podría no ser el mismo que el receptor finalmente recibe, ya que se usan zonas de memoria, accesibles desde los procesos involucrados, que podrían escribirse cuando el mensaje está en tránsito.

- **Operación de envío-recepción segura**: Ocurre cuando se puede garantizar a priori que el valor de `a` en E_1 coincidirá con el valor de `b` justo tras R_3.
- **Operación de envío-recepción insegura**: Una operación puede ser no segura en dos circunstancias:
  - **Envío inseguro:** ocurre cuando es posible modificar el valor de `a` entre E_1 y E_3 (podría enviarse un valor distinto del valor en E_1).
  - **Recepción insegura:** ocurre cuando se puede acceder a `b` entre R_1 y R_3, si se lee antes de recibirlo totalmente o se modifica después de haberse ya recibido parcialmente.

Tipos de operaciones de paso de mensajes

Operaciones seguras

- Devuelven el control cuando se garantiza la seguridad.
- No siempre significa que el receptor haya recibido el dato cuando finalice el **`send`**, sino que cambiar los datos no viola la seguridad.
- Existen 2 mecanismos de paso de mensajes seguro:
  - **Envío y recepción síncronos**.
  - **Envío asíncrono seguro**

Operaciones inseguras

- Devuelven el control de forma inmediata sin garantizar la seguridad.
- Es responsabilidad del usuario asegurar que no se alteran los datos mientras el mensaje está en tránsito.
- Deben existir sentencias adicionales para comprobar el estado de la operación.

El estándar **MPI** incluye funciones con todas estos comportamientos.

Operaciones síncronas. Comportamiento.

```
s_send( variable, id_proceso_receptor ) ;
```

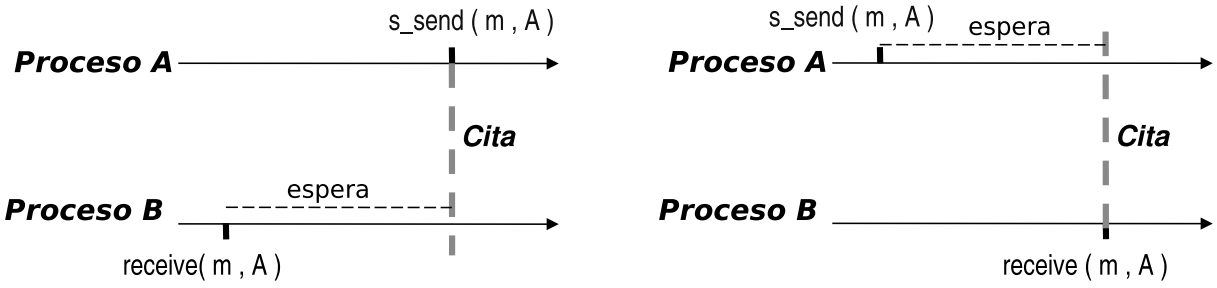
- Realiza el envío de los datos y espera bloqueado hasta que los datos hayan terminado de leerse en el emisor y se haya iniciado el **`receive`** en el receptor.
- Es decir: devuelve el control después de E_3 y de R_1 (después del posterior de los dos).

```
receive( variable, id_proceso_emisor ) ;
```

- Espera bloqueado hasta que el emisor emita un mensaje con destino al proceso receptor (si no lo había hecho ya), y hasta que hayan terminado de escribirse los datos en la zona de memoria designada en *variable*.
- Es decir: devuelve el control después de R_3.

Operaciones síncronas. Cita.

- Exige **cita** entre emisor y receptor: La operación **`s_send`** no devuelve el control hasta que el **`receive`** correspondiente sea alcanzado en el receptor
  - El intercambio de mensaje constituye un punto de sincronización entre emisor y receptor.
  - El emisor podrá hacer aserciones acerca del estado del receptor.
  - Análogo: comunicación telefónica y chat.

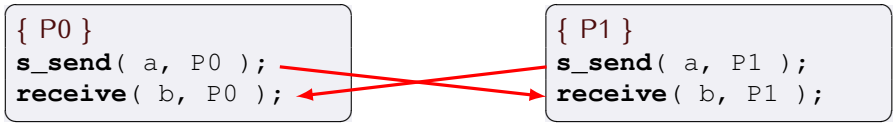


Operaciones síncronas. Desventajas.

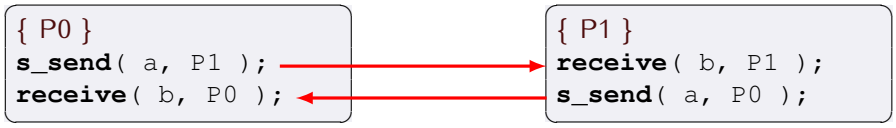
- Fácil de implementar pero poco flexible.
- **Sobrecarga por espera ociosa**: adecuado sólo cuando send/receive se inician aprox. mismo tiempo.

- **Interbloqueo**: es necesario alternar llamadas en intercambios (código menos legible).

Ejemplo de Interbloqueo



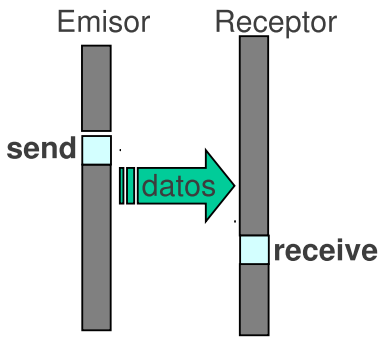
Corrección



Envío asíncrono seguro (1)

```
send( variable, id_proceso_receptor ) ;
```

- Inicia el envío de los datos designados y espera bloqueado hasta que hayan terminado de copiarse todos los datos de `variable` a un lugar seguro. Tras la copia de los datos designados, devuelve el control sin que tengan que haberse recibido los datos en el receptor.
- Por tanto, devuelve el control después de E_3.
- Se suele usar junto con la recepción síncrona (**`receive`**).



Envío asíncrono seguro. Ventajas e Inconvenientes

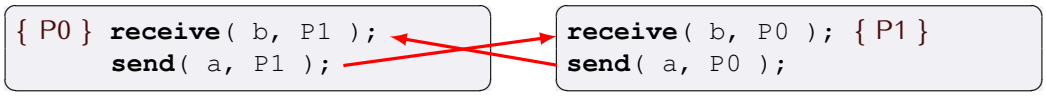
- Alivia sobrecargas de espera ociosa a costa de gestión de almacenamiento temporal.
- Sólo menos ventajosas en programas altamente síncronos o cuando la capacidad del almacenamiento temporal sea un asunto crítico.
- **Impacto del almacenamiento intermedio finito**: Se deben escribir programas con requisitos de almacenamiento acotados.

- Si $P1$ es más lento que $P0$, $P0$ podría continuar solo mientras exista espacio de almacenamiento temporal.

```
{ P0 } for i:=1 to 10000 do begin
          a := Produce() ;
          send( a, P1 ) ;
       end
```

```
{ P1 } for i:=1 to 10000 do begin
          receive( a, P0 ) ;
          Consume( a ) ;
       end
```

- **Posibilidad de Interbloqueo**: No olvidar que las llamadas a `receive` siguen siendo síncronas.



Operaciones inseguras

- **Operaciones seguras**: Garantizan la seguridad, pero presentan ineficiencias:
  - sobrecarga de espera ociosa (operaciones síncronas).
  - sobrecarga de gestión de almacenamiento temporal (envío asíncrono seguro).
- **Alternativa**: Requerir que el programador asegure la corrección y usar operaciones `send/receive` con baja sobrecarga.
  - Las operaciones devuelven el control antes de que sea seguro modificar (en el caso del envío) o leer los datos (en el caso de la recepción).
  - Es responsabilidad del usuario asegurar que el comportamiento es correcto.
  - Deben existir **sentencias de chequeo de estado**: indican si los datos pueden alterarse o leerse sin comprometer la seguridad.
  - Una vez iniciada la operación, el usuario puede realizar cualquier cómputo que no dependa de la finalización de la operación y, cuando sea necesario, chequeará el estado de la operación.

Paso de mensajes asíncrono inseguro. Operaciones

i_send(*variable*, *id_proceso_receptor*, *var_resguardo*) ;

- Indica al SPM que comienze una operación de envío al proceso receptor designado. No espera a que terminen de leerse los bytes del emisor, ni a que terminen de escribirse en el receptor.
- Es decir: devuelve el control después de E_1.
- *var_resguardo* permite consultar el estado del envío.

i_receive (*variable*, *id_proceso_receptor*, *var_resguardo*) ;

- Indica al SPM que el proceso receptor desea recibir en la zona de memoria local designada en *variable* un mensaje del proceso emisor indicado. Se devuelve el control sin esperar a que se complete el envío ni a que llegen los datos.
- Es decir: devuelve el control después de R_1.
- *var_resguardo* permite consultar después el estado de la recepción.

Esperar el final de operaciones asíncronas

Cuando un proceso hace **i_send** o **i_receive** puede continuar trabajando hasta que llega un momento en el que debe esperar a que termine la operación. Para esto se disponen de estos dos procedimientos:

wait_send(*var_resguardo*) ;

- Se invoca por el proceso emisor, y lo bloquea hasta que la operación de envío asociada a *var_resguardo* ha llegado al instante E_3.

wait_recv (*var_resguardo*) ;

- Se invoca por el proceso receptor, que queda bloqueado hasta que la operación de recepción asociada a *var_resguardo* ha llegado al instante R_3.

Paso de mensajes asíncrono inseguro (no bloqueante)

- El proceso que ejecuta **i_send** informa al SPM de un mensaje pendiente y continúa.
  - El programa puede hacer mientras otro trabajo (`trabajo_util`)
  - El SPM iniciará la comunicación cuando corresponda.
  - Operaciones de chequeo indican si es seguro tocar/leer los datos.
- **Mejora**: El tiempo de espera ociosa se puede emplear en computación
- **Coste**: Reestructuración programa, mayor esfuerzo del programador.

```
process Emisor ;
   var a : integer := 100 ;
begin
  i_send( a, Receptor, resg );
  { trabajo util : no escribe en 'a'}
  trabajo_util_emisor();
  wait_send(resg);
  a := 0 ;
end
```

```
process Receptor ;
   var b : integer ;
begin
 i_receive( b, Emisor, resg );
 { trabajo util : no accede a 'b'}
 trabajo_util_receptor();
 wait_recv (resg);
 imprime( b );
end
```

Ejemplo: Productor-Consumidor con paso asíncrono

Se logra simultanear transmisión, producción y consumición:

```
process Productor ;
   var x, x_env : integer ;
begin
 x := Producir()
 while true do begin
   x_env := x ;
   i_send(x_env,Consumidor,resg);
   x := Producir() ;
   wait_send(resg);
 end
end
```

```
process Consumidor ;
   var y, y_rec : integer ;
begin
 i_receive(y_rec,Productor,resg);
 while true do begin
   wait_recv(resg);
   y := y_rec ;
   i_receive(y_rec,Productor,resg);
   Consumir(y);
 end
end
```

Limitaciones:

- La duración del trabajo util podría ser muy distinta de la de cada transmisión.
- Se descarta la posibilidad de esperar más de un posible emisor.

Consulta de estado de operaciones asíncronas

Para solventar los dos problemas descritos, se pueden usar dos funciones de comprobación de estado de una transmisión de un mensaje. No suponen bloqueo, solo una consulta:

test_send(*var_resguardo*) ;

- Función lógica que se invoca por el emisor. Si el envío asociado a *var_resguardo* ha llegado a E_3, devuelve **`true`**, sino devuelve **`false`**.

test_recv (*var_resguardo*) ;

- Función lógica que se invoca por el receptor. Si el envío asociado a *var_resguardo* ha llegado a R_3, devuelve **`true`**, sino devuelve **`false`**.

Paso asíncrono con espera ocupada posterior.

El trabajo util de nuevo se puede simultánear con la transmisión del mensaje, pero en este caso dicho debe descomponerse en muchas tareas pequeñas.

```
process Emisor ;
   var a : integer := 100 ;
begin
  i_send( a, Receptor, resg );
  while not test_send( resg ) do
    { trabajo util: no escribe en 'a'}
    trabajo_util_emisor();
  a := 0 ;
end
```

```
process Receptor ;
   var b : integer ;
begin
 i_receive( b, Emisor, resg );
 while not test_recv( resg ) do
   { trabajo util: no accede a 'b'}
   trabajo_util_receptor();
 imprime( b );
end
```

Si el trabajo util es muy corto, puede convertirse en una espera ocupada que consume muchísima CPU inútilmente. Es complejo de ajustar bien.

Recepción simultánea de varios emisores.

En este caso se comprueba continuamente si se ha recibido un mensaje de uno cualquiera de dos emisores, y se espera hasta que se han recibido de los dos:

```
process Emisor1 ;
var a :integer:= 100;
begin
  send(a,Receptor);
end

process Emisor2 ;
var b :integer:= 200;
begin
  send(b,Receptor);
end
```

```
process Receptor ;
   var b1, b2 : integer ;
       r1,r2 : boolean := false ;
begin
 i_receive( b1, Emisor1, resg1 );
 i_receive( b2, Emisor2, resg2 );
 while not r1 or not r2 do begin
   if not r1 and test_recv( resg1 ) then begin
      r1 := true ;
      imprime(" recibido de 1 : ", b1 );
   end
   if not r2 and test_recv( resg2 ) then begin
      r2 := true ;
      imprime(" recibido de 2 : ", b2 );
   end
 end
end
```

4.1.4 Espera selectiva

Espera Selectiva. Introducción al problema

- Los modelos basados en paso de mensajes imponen ciertas restricciones.
- No basta con **send** y **receive** para modelar el comportamiento deseado.

Productor-Consumidor con buffer de tamaño fijo

- Se asumen operaciones síncronas.

- **Problema**: Acoplamiento forzado entre los procesos.

```
process Productor ;
begin
   while true do begin
      v := Produce();
      s_send(v,Consumidor);
   end
end
```

```
process Consumidor ;
begin
   while true do begin
      receive(v,Productor);
      Consume(v);
   end
end
```

Introducción al problema. Mejora

- **Mejora**: Gestión intercambio mediante proceso Intermedio.
  - Productor puede continuar después envío.
  - **Problema**: El proceso `Intermedio` sólo puede esperar mensajes de un único emisor en cada instante.
- Aspecto común en aplicaciones cliente-servidor.
  - No se conoce a priori el cliente que hace la petición en cada instante.
  - Servidor debe estar preparado para recibir sin importar orden.

```
{ Productor (P) }
process P ;
begin
 while true do begin
  v:=Produce();
  s_send(v,B);
 end
end
```

```
{ Intermedio (B) }
process B ;
begin
 while true do begin
  receive(v,P);
  receive(s,C);
  s_send(v,C);
 end
end
```

```
{ Consumidor (C) }
process C ;
begin
 while true do begin
  s_send(s,B);
  receive(v,B);
  Consume(v);
 end
end
```

Espera selectiva con alternativas guardadas (1)

Solución: usar **espera selectiva** con varias alternativas guardadas. Es una nueva sentencia compuesta, con la siguiente sintaxis:

```
select
   when condicion_1 receive( variable_1, proceso_1 ) do
      sentencias_1
   when condicion_2 receive( variable_2, proceso_2 ) do
      sentencias_2
   ...
   when condicion_n receive( variable_n, proceso_n ) do
      sentencias_n
end
```

- Cada bloque que comienza en **`when`** se llama una **alternativa**.
- En cada alternativa, el texto desde **`when`** hasta **`do`** se llama la **guarda** de dicha alternativa, la guarda puede incluir una expresión lógica ($condicion_i$)
- Los **`receive`** nombran a otros procesos del programa concurrente ($proceso_i$), y cada uno referencia una variable local ($variable_i$), donde eventualmente se recibirá un valor del proceso asociado.

Sintaxis de las guardas.

En una guarda de una orden **`select`**:

- La expresión lógica puede omitirse, es ese caso la sintaxis sería:

```
when receive( mensaje, proceso ) do
   sentencias
```

  que es equivalente a:

```
when true receive( mensaje, proceso ) do
      sentencias
```

- La sentencia **receive** puede no aparecer, la sintaxis es:

```
when condicion do
     sentencias
```

  decimos que esta es una guarda sin sentencia de entrada.

Guardas ejecutables. Evaluación de las guardas.

Una guarda es **ejecutable** en un momento de la ejecución de un proceso P cuando se dan estas dos condiciones:

- La condición de la guarda se evalúa en ese momento a **`true`**.
- Si tiene sentencia de entrada, entonces el proceso origen nombrado ya ha iniciado en ese momento una sentencia **`send`** (de cualquier tipo) con destino al proceso P, que casa con el **`receive`**.

Una guarda será **potencialmente ejecutable** si se dan estas dos condiciones:

- La condición de la guarda se evalúa a **`true`**.
- Tiene una sentencia de entrada, sentencia que nombra a un proceso que no ha iniciado aún un **`send`** hacia P.

Una guarda será **no ejecutable** en el resto de los casos, en los que forzosamente la condición de la guarda se evalúa a **`false`**.

Evaluación de las guardas.

Cuando el flujo de control llega a un **`select`**, se evalúan las condiciones de todas las guardas, y se verifica el estado de todos los procesos nombrados en los **`receive`**. De esta forma se clasifican las guardas, y se selecciona una alternativa:

- Si hay guardas ejecutables con sentencia de entrada: se selecciona aquella cuyo **`send`** se inició antes (esto garantiza a veces la equidad).
- Si hay guardas ejecutables, pero ninguna tiene una sentencia de entrada: se selecciona aleatoriamente una cualquiera.
- Si no hay ninguna guarda ejecutable, pero sí hay guardas potencialmente ejecutables: se espera (bloqueado) a que alguno de los procesos nombrados en esas guardas inicie un **`send`**, en ese momento acaba la espera y se selecciona la guarda correspondiente a ese proceso.
- Si no hay guardas ejecutables ni potencialmente ejecutables: no se selecciona ninguna guarda.

Ejecución de un select

Para ejecutar un select primero se intenta seleccionar una guarda de acuerdo con el esquema descrito arriba.

- Si no se puede seleccionar ninguna guarda, no se hace nada (no hay guardas ni ejecutables, ni potencialmente ejecutables).
- Si se ha podido seleccionar una guarda, entonces se dan estos dos pasos en secuencia:

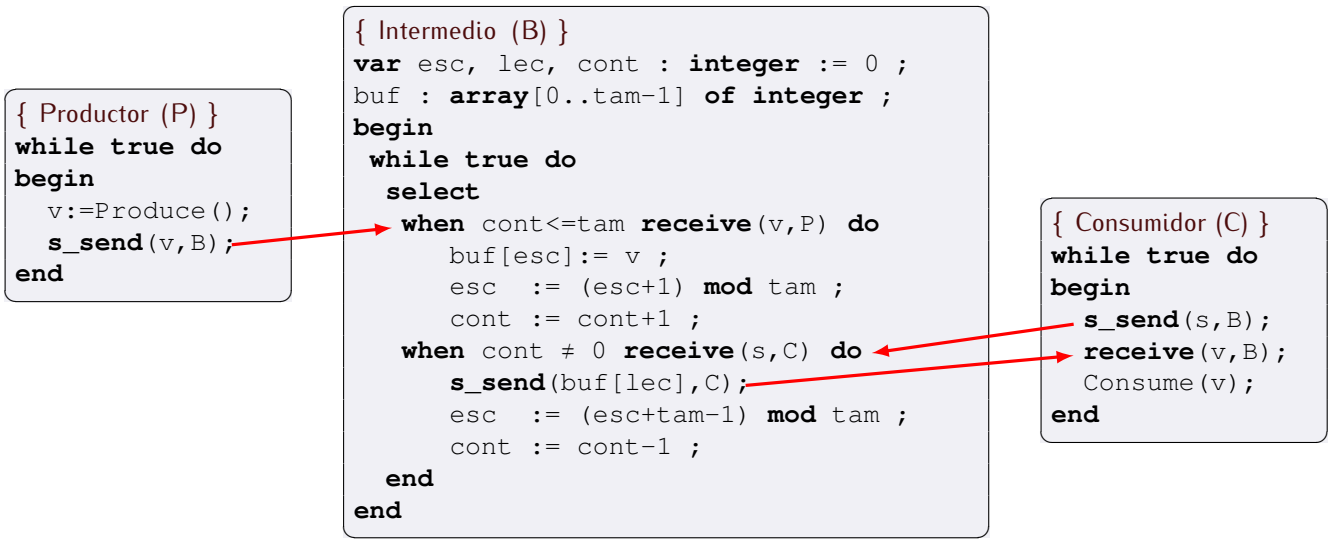
  1. Si esa guarda tiene sentencia de entrada, se ejecuta el **`receive`** (siempre habrá un **`send`** iniciado), y se recibe el mensaje.
  2. Se ejecuta la sentencia asociada a la alternativa.

  y después finaliza la ejecución del select.

Hay que tener en cuenta que **`select`** conlleva potencialmente esperas, y por tanto se pueden producir esperas indefinidas (interbloqueo).
Existe **`select`** con prioridad: la selección no sería arbitraria (el orden en el que aparecen las alternativas establece la prioridad).

Productor-Consumidor con buffer limitado

- Intermedio no conoce a priori orden de peticiones (Inserción/Extracción).
- Las guardas controlan las condiciones de sincronización (seguridad).



Select con guardas indexadas

A veces es necesario replicar una alternativa. En estos casos se puede usar una sintaxis que evita reescribir el código muchas veces, con esta sintaxis:

```
for indice := inicial to final
   when condicion receive( mensaje, proceso ) do
      sentencias
```

Tanto la *condición*, como el *mensaje*, el *proceso* o la *sentencia* pueden contener referencias a la variable *indice* (usualmente es `i` o `j`). Es equivalente a:

```
when condicion receive( mensaje, proceso ) do
      sentencias { se sustituye indice por inicial }
when condicion receive( mensaje, proceso ) do
      sentencias { se sustituye indice por inicial + 1 }
...
when condicion receive( mensaje, proceso ) do
      sentencias { se sustituye indice por final }
```

Ejemplo de select con guardas indexadas

A modo de ejemplo, si `suma` es un vector de n enteros, y `fuente[0]`, `fuente[1]`, etc... son n procesos, entonces:

```
for i := 0 to n-1
   when suma[i] < 1000 receive( numero, fuente[i] ) do
      suma[i] := suma[i] + numero ;
```

Es equivalente a:

```
when suma[0] < 1000 receive( numero, fuente[0] ) do
   suma[0] := suma[0] + numero ;
when suma[1] < 1000 receive( numero, fuente[1] ) do
```

```
   suma[1] := suma[1] + numero ;
...
when suma[n-1] < 1000 receive( numero, fuente[n-1] ) do
   suma[n-1] := suma[n-1] + numero ;
```

En un **select** se pueden combinar una o varias alternativas indexadas con alternativas normales no indexadas.

Ejemplo de select

Este ejemplo suma los primeros numeros recibidos de cada uno de los n procesos fuente hasta que cada suma iguala o supera al valor 1000:

```
var suma      : array[0..n-1] of integer := (0,0,...,0) ;
    continuar : boolean := false ;
    numero    : integer ;
begin
   while continuar do begin
      continuar := false ; { terminar cuando ∀i suma[i] ≥ 1000 }
      select
         for i := 0 to n-1
         when suma[i] < 1000 receive( numero, fuente[i] ) do
            suma[i] := suma[i]+numero ; { sumar }
            continuar := true ; { iterar de nuevo }
      end
   end
end
```

aquí hemos supuesto que los procesos `fuente` están definidos como:

```
process fuente[ i : 0..n-1 ] ;
begin
   .....
end
```

4.2 Paradigmas de interacción de procesos en programas distribuidos

4.2.1 Introducción

Introducción

Paradigma de interacción

Un **paradigma** de interacción define un esquema de interacción entre procesos y una estructura de control que aparece en múltiples programas.

- Unos pocos paradigmas de interacción se utilizan repetidamente para desarrollar muchos programas distribuidos.
- Veremos los siguientes paradigmas de interacción:

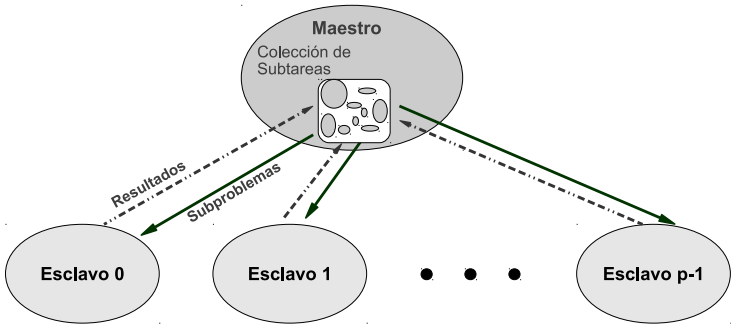
1. Maestro-Esclavo.
2. Iteración síncrona.
3. Encauzamiento (pipelining).
4. Cliente-Servidor.

- Se usan principalmente en programación paralela, excepto el ultimo que es más general (sistemas distribuidos) y se verá en el siguiente apartado del capítulo (Mecanismos de alto nivel para paso de mensajes).

4.2.2 Maestro-Esclavo

Maestro-Esclavo

- En este patrón de interacción intervienen dos entidades: un proceso maestro y múltiples procesos esclavos.
- El **proceso maestro** descompone el problema en pequeñas subtareas (que guarda en una colección), las distribuye entre los procesos esclavos y va recibiendo los resultados parciales de estos, de cara a producir el resultado final.
- Los **procesos esclavos** ejecutan un ciclo muy simple hasta que el maestro informa del final del cómputo: reciben un mensaje con la tarea , procesan la tarea y envían el resultado al maestro.

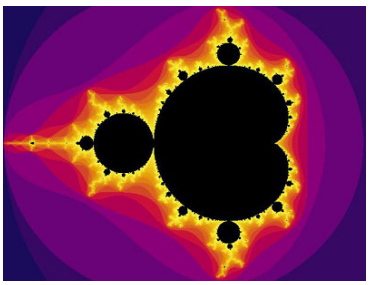


Ejemplo: Cálculo del Conjunto de Mandelbrot

- **Conjunto de Mandelbrot**: Conjunto de puntos c del plano complejo (dentro de un círculo de radio 2 centrado en el origen) que no excederán cierto límite cuando se calculan realizando la siguiente iteración (inicialmente $z = 0$ con $z = a + bi \in \mathbb{C}$):

$$\text{Repetir } z_{k+1} := {z_k}^2 + c \quad \text{hasta} \quad \|z\| > 2 \text{ o } k > \textit{límite}$$

- Se asocia a cada pixel (con centro en el punto c) un color en función del número de iteraciones (k) necesarias para su cálculo.
- **Conjunto solución**= {pixels que agoten iteraciones límite dentro de un círculo de radio 2 centrado en el origen}.



Ejemplo: Cálculo del Conjunto de Mandelbrot

- **Paralelización sencilla**: Cada pixel se puede calcular sin ninguna información del resto
- **Primera aproximación**: asignar un número de pixels fijo a cada proceso esclavo y recibir resultados.
  - **Problema**: Algunos procesos esclavos tendrían más trabajo que otros (el número de iteraciones por pixel no es fijo) para cada pixel.
- **Segunda aproximación**:
  - El maestro tiene asociada una colección de filas de pixels.
  - Cuando los procesos esclavos están ociosos esperan recibir una fila de pixels.
  - Cuando no quedan más filas, el Maestro espera a que todos los procesos completen sus tareas pendientes e informa de la finalización del cálculo.
- Veremos una solución que usa envío asíncrono seguro y recepción síncrona.

Procesos Maestro y Esclavo

```
process Maestro ;
begin
   for i := 0 to num_esclavos-1 do
      send( fila, Esclavo[i] ) ;      { enviar trabajo a esclavo   }
   while queden filas sin colorear do
      select
         for j := 0 to n_e - 1 when receive( colores, Esclavo[j] ) do
            if quedan filas en la bolsa
               then send( fila, Esclavo[j] )
               else send( fin, Esclavo[j] );
            visualiza(colores);
      end
end
```

```
process Esclavo[ i : 0..num_esclavos-1 ] ;
begin
   receive( mensaje, Maestro );
   while mensaje != fin do begin
      colores := calcula_colores(mensaje.fila) ;
      send (colores, Maestro );
```

```
      receive( mensaje, Maestro );
   end
end
```

4.2.3 Iteración síncrona

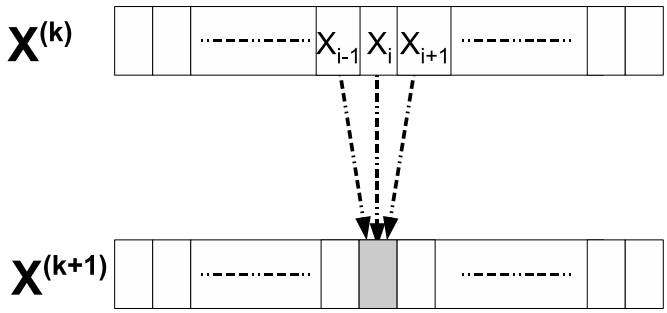
Iteración síncrona

- **Iteración**: En múltiples problemas numéricos, un cálculo se repite y cada vez se obtiene un resultado que se utiliza en el siguiente cálculo. El proceso se repite hasta obtener los resultados deseados.
- A menudo se pueden realizar los cálculos de cada iteración de forma concurrente.
- **Paradigma de iteración síncrona**:
  - En un bucle diversos procesos comienzan juntos en el inicio de cada iteración.
  - La siguiente iteración no puede comenzar hasta que todos los procesos hayan acabado la previa.
  - Los procesos suelen intercambiar información en cada iteración.

Ejemplo: Transformación iterativa de un vector (1)

Supongamos que debemos realizar M iteraciones de un cálculo que transforma un vector x de n reales:

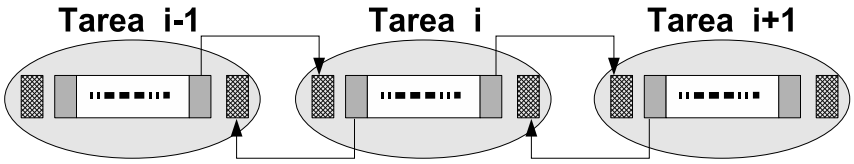
$$x_i^{(k+1)} = \frac{x_{i-1}^{(k)} - x_i^{(k)} + x_{i+1}^{(k)}}{2}, \qquad i = 0,\ldots,n-1, \quad k = 0,1,\ldots,N,$$
$$x_{-1}^{(k)} = x_{n-1}^{(k)}, \qquad x_n^{(k)} = x_0^{(k)}.$$



Veremos una solución que usa envío asíncrono seguro y recepción síncrona.

Ejemplo: Transformación iterativa de un vector (2)

El esquema que se usará para implementar será este:



- El número de iteraciones es una constante predefinida m
- Se lanzan p procesos concurrentes.
- Cada proceso guarda una parte del vector completo, esa parte es un vector local con n/p entradas reales, indexadas de 0 a $n/p-1$ (vector `bloque`) (asumimos que n es múltiplo de p)
- Cada proceso, al inicio de cada iteración, se comunica con sus dos vecinos las entradas primera y última de su bloque.
- Al inicio de cada proceso, se leen los valores iniciales de un vector compartido que se llama `valores` (con n entradas), al final se copian los resultados en dicho vector.

Ejemplo: Transformación iterativa de un vector (3)

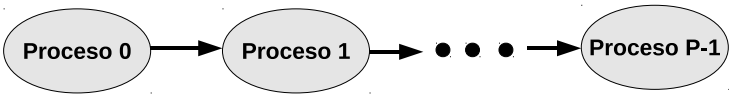
El código de cada proceso puede quedar así:

```
process Tarea[ i : 0..p-1 ] ;
   var bloque : array[0..n/p-1] of float ; { bloque local (no compartido) }
begin
   for j := 0 to n/p-1 do bloque[j] := valores[i*n/p+j] ; { lee valores }
   for k := 0 to m do begin { bucle que ejecuta las iteraciones }
      { comunicacion de valores extremos con los vecinos }
      send( bloque[0]     ,Tarea[i-1 mod p] ); { enviar primero a anterior }
      send( bloque[n/p-1],Tarea[i+1 mod p] ); { enviar ultimo a siguiente }
      receive( izquierda, Tarea[i-1 mod p] ); { recibir ultimo de anterior }
      receive( derecha,   Tarea[i+1 mod p] ); { recibir primero del siguiente }
      { calcular todas las entradas excepto la ultima }
      for j := 0 to n/p-2 do begin
         tmp := bloque[j] ;
         bloque[j] := ( izquierda - bloque[j] + bloque[j+1] )/2;
         izquierda := tmp ;
      end
      { calcular ultima entrada }
      bloque[n/p-1] := ( izquierda - bloque[n/p-1] + derecha )/2;
   end
   for j := 0 to n/p-1 do valores[i*n/p+j] := bloque[j] ; { escribre resultados }
end
```

4.2.4 Encauzamiento (pipelining)

Encauzamiento (pipelining)

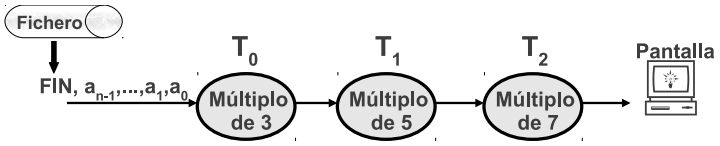
- El problema se divide en una serie de tareas que se han de completar una después de otra
- Cada tarea se ejecuta por un proceso separado.
- Los procesos se organizan en un cauce (pipeline) donde cada proceso se corresponde con una *etapa* del cauce y es responsable de una tarea particular.
- Cada etapa del cauce contribuirá al problema global y devuelve información que es necesaria para etapas posteriores del cauce.
- Patrón de comunicación muy simple ya que se establece un flujo de datos entre las tareas adyacentes en el cauce.



Encauzamiento: Ejemplo (1)

Cauce paralelo para filtrar una lista de enteros

- Dada una serie de m primos $p_0, p_1, ..., p_{m-1}$ y una lista de n enteros, $a_0, a_1, a_2, ..., a_{n-1}$, encontrar aquellos números de la lista que son múltiplos de todos los m primos ($n >> m$)
- El proceso `Etapa[`i`]` (con $i = 0, \ldots, m-1$) mantiene el primo p_i y chequea multiplicidad con p_i.
- Veremos una solución que usa operaciones síncronas.



Encauzamiento: Ejemplo (2)

```
{ vector (compartido) con la lista de primos }
var primos : array[0..m-1] of float := { p0, p1, p2, ···, pm-1 } ;

{ procesos que ejecutan cada etapa: }

process Etapa[ i : 0..m-1 ] ;
   var izquierda : integer := 0 ;
begin
   while izquierda >= 0 do begin
      if i == 0 then
         leer( izquierda ); { obtiene siguiente entero }
      else
         receive( izquierda, Etapa[i-1]);
      if izquierda mod primos[i] == 0 then begin
         if i != m-1 then
            s_send ( izquierda, Etapa[i+1]);
         else
            imprime( izquierda );
      end
   end
end
```

4.3 Mecanismos de alto nivel en sistemas distribuidos

4.3.1 Introducción

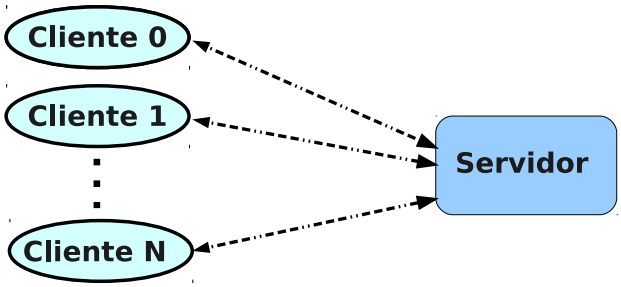
Introducción

- Los mecanismos vistos hasta ahora (varios tipos de envío/recepción espera selectiva, ...) presentan un bajo nivel de abstracción.
- Se verán dos mecanismos de mayor nivel de abstracción: **Llamada a procedimiento remoto (RPC)** e **Invocación remota de métodos (RMI)**.
- Están basados en la forma de comunicación habitual en un programa: *llamada a procedimiento*.
  - El llamador proporciona nombre del proc + parámetros, y espera.
  - Cuando proc. termina, el llamador obtiene los resultados y continúa.
- En el **modelo de invocación remota**:
  - El llamador invoca desde un proceso o máquina diferente de donde se encuentra el procedimiento invocado.
  - El llamador se queda bloqueado hasta que recibe los resultados (esquema síncrono).
  - El flujo de comunicación es bidireccional (petición-respuesta).
  - Se permite que varios procesos invoquen un procedimiento gestionado por otro proceso (esquema muchos a uno).

4.3.2 El paradigma Cliente-Servidor

El paradigma Cliente-Servidor

- Paradigma más frecuente en programación distribuida.
- Relación asimétrica entre dos procesos: cliente y servidor.
  - **Proceso servidor**: gestiona un recurso (por ejemplo, una base de datos) y ofrece un servicio a otros procesos (clientes) para permitir que puedan acceder al recurso. Puede estar ejecutándose durante un largo periodo de tiempo, pero no hace nada útil mientras espera peticiones de los clientes.
  - **Proceso cliente**: necesita el servicio y envía un mensaje de petición al servidor solicitando algo asociado al servicio proporcionado por el servidor (p.e. una consulta sobre la base de datos).



El paradigma Cliente-Servidor

Es sencillo implementar esquemas de interacción cliente-servidor usando los mecanismos vistos. Para ello usamos en el servidor un **select** que acepta peticiones de cada uno de los clientes:

```
process Cliente[ i : 0..n-1 ] ;
begin
   while true do begin
      s_send( peticion, Servidor );
      receive( respuesta, Servidor );
   end
end

process Servidor ;
begin
 while true do
   select
      for i:= 0 to n-1
      when condicion[i] receive( peticion, Cliente[i] ) do
         respuesta := servicio( peticion ) ;
         s_send( respuesta, Cliente[i] ),
   end
end
```

Problemas de la solución

No obstante, se plantean problemas de seguridad en esta solución:

- Si el servidor falla, el cliente se queda esperando una respuesta que nunca llegará.
- Si un cliente no invoca el **`receive`**`(respuesta,Servidor)` y el servidor realiza **`s_send`**`(respuesta,Cliente[j]` síncrono, el servidor quedará bloqueado

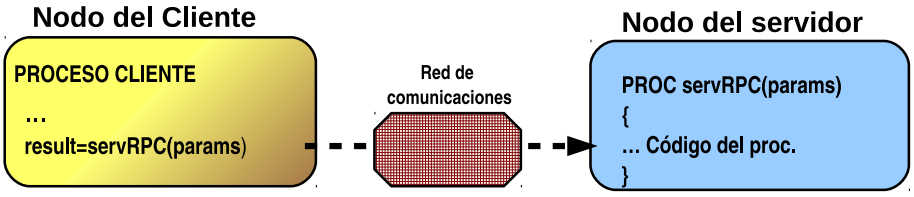
Para resolver estos problemas:

- El par (recepción de petición, envío de respuesta) se debe considerar como una única operación de comunicación bidireccional en el servidor y no como dos operaciones separadas.
- El mecanismo de **llamada a procedimiento remoto** (RPC) proporciona una solución en esta línea.

4.3.3 Llamada a Procedimiento (RPC)

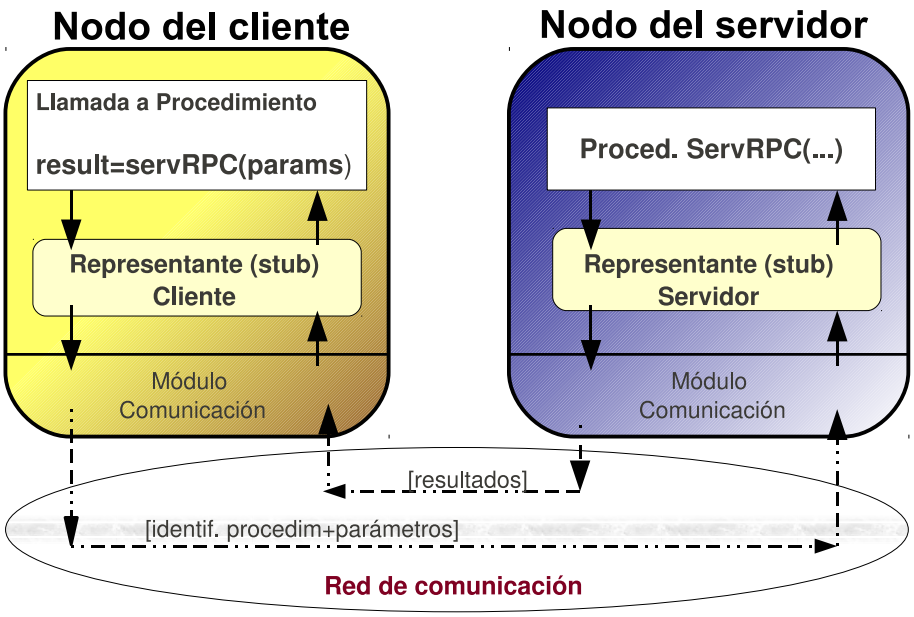
Introducción a RPC

- **Llamada a procedimiento remoto (Remote Procedure Call)**: Mecanismo de comunicación entre procesos que sigue el esquema cliente-servidor y que permite realizar las comunicaciones como llamadas a procedimientos convencionales (locales).
- **Diferencia ppal respecto a una llamada a procedimiento local**: El programa que invoca el procedimiento (cliente) y el procedimiento invocado (que corre en un proceso servidor) pueden pertenecer a máquinas diferentes del sistema distribuido.



Esquema de interacción en una RPC

- **Representante o delegado (stub)**: procedimiento local que gestiona la comunicación en el lado del cliente o del servidor.
- Los procesos cliente y servidor no se comunican directamente, sino a través de representantes.



Esquema general de una RPC. Inicio en el nodo cliente

1. En el nodo cliente se invoca un procedimiento remoto como si se tratara de una llamada a procedimiento local. Esta llamada se traduce en una llamada al *representante* del cliente.
2. El representante del cliente empaqueta todos los datos de la llamada (nombre del procedimiento y parámetros) usando un determinado formato para formar el cuerpo del mensaje a enviar (es muy usual utilizar el proticolo XDR, eXternal Data Representation). Este proceso se suele denominar **marshalling** o **serialización**.
3. El representante el cliente envía el mensaje con la petición de servicio al nodo servidor usando el módulo de comunicación del sistema operativo.
4. El programa del cliente se quedará bloqueado esperando la respuesta.

Esquema general de una RPC. Pasos en el nodo servidor y recepción de resultados en cliente

1. En el nodo servidor, el sistema operativo desbloquea al proceso servidor para que se haga cargo de la petición y el mensaje es pasado al representante del servidor.
2. El representante del servidor desempaqueta (*unmarshalling*) los datos del mensaje de petición (identificación del procedimiento y parámetros) y ejecuta una llamada al procedimiento local identificado usando los parámetros obtenidos del mensaje.
3. Una vez finalizada la llamada, el representante del servidor empaqueta los resultados en un mensaje y lo envía al cliente.
4. El sistema operativo del nodo cliente desbloquea al proceso que hizo la llamada para recibir el resultado que es pasado al representante del cliente.

5. El representante del cliente desempaqueta el mensaje y pasa los resultados al invocador del procedimiento.

Representación de datos y paso de parámetros en la RPC

- **Representación de los datos**
  - En un sistema distribuido los nodos pueden tener diferente hardware y/o sistema operativo (sistema heterogéneo), utilizando diferentes formatos para representar los datos.
  - En estos casos los mensajes se envían usando una representación intermedia y los representantes de cliente y servidor se encargan de las conversiones necesarias.
- **Paso de parámetros** En RPC, los parámetros de la llamada se pueden pasar:
  - *por valor*: en este caso basta con enviar al representante del servidor los datos aportados por el cliente.
  - *por referencia*: En este caso se pasa un puntero, por lo que se han de transferir también los datos referenciados al servidor. Además, en este caso, cuando el procedimiento remoto finaliza, el representante del servidor debe enviar al cliente, junto con los resultados, los parámetros pasados por referencia que han sido modificados.

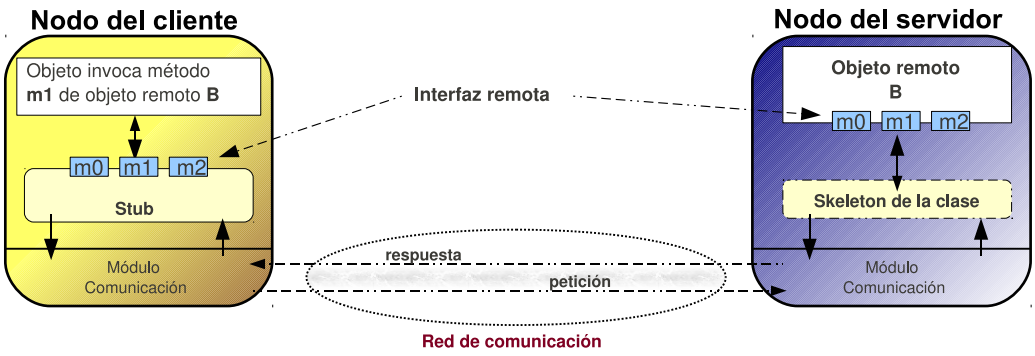
4.3.4 Java Remote Method Invocation (RMI)

El modelo de objetos distribuidos

- **Invocación de métodos en progr. orientada a objetos**
  - En programación orientada a objetos, los objetos se comunican entre sí mediante invocación a métodos.
  - Para invocar el método de un objeto hay que dar una referencia del objeto, el método concreto y los argumentos de la llamada.
  - La interfaz de un objeto define sus métodos, argumentos, tipos de valores devueltos y excepciones.
- **Invocación de métodos remotos**
  - En un entorno distribuido, un objeto podría invocar métodos de un objeto localizado en un nodo o proceso diferente del sistema (**objeto remoto**) siguiendo el paradigma cliente-servidor como ocurre en RPC.
  - Un objeto remoto podría recibir invocaciones locales o remotas. Para invocar métodos de un objeto remoto, el objeto que invoca debe disponer de la *referencia* del objeto remoto del nodo receptor, que es única en el sistema distribuido.

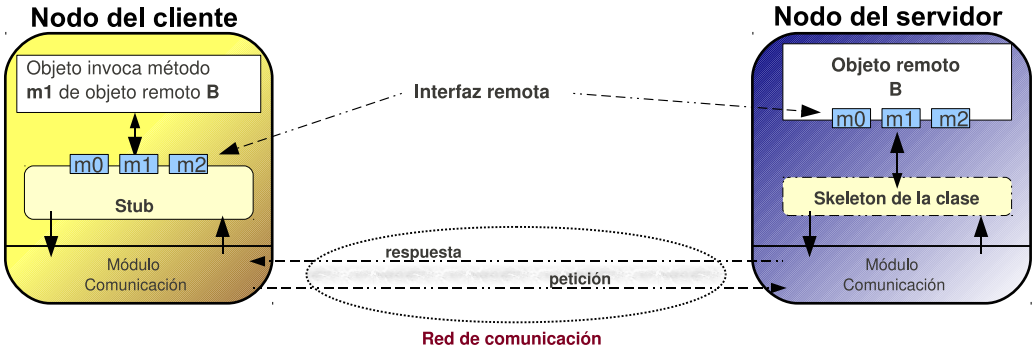
El modelo de objetos distribuidos. Interfaz remota y representantes

- **Interfaz remota**: especifica los métodos del objeto remoto que están accesibles para los demás objetos así como las excepciones derivadas (p.e., que el servidor tarde mucho en responder).
- **Remote Method Invocation (RMI)**: acción de invocar un método de la interfaz remota de un objeto remoto. La invocación de un método en una interfaz remota sigue la misma sintaxis que un objeto local.



El modelo de objetos distribuidos. Interfaz remota y representantes

- El esquema de RMI es similar a la RPC:
  - **En el cliente**: un representante local de la clase del objeto receptor (*stub*) implementa el marshalling y la comunicación con el servidor, compartiendo la misma interfaz que el objeto receptor.
  - **En el servidor**: la implementación de la clase del objeto receptor (*skeleton*) recibe y traduce las peticiones del stub, las envía al objeto que implementa la interfaz remota y espera resultados para formatearlos y enviárselos al stub del cliente.



El modelo de objetos distribuidos. Referencias remotas

- Los stubs se generan a partir de la definición de la interfaz remota.
- Los objetos remotos residen en el nodo servidor y son gestionados por el mismo.

- Los procesos clientes manejan **referencias remotas** a esos objetos.
  - Obtienen una referencia unívoca con la localización del objeto remoto dentro del sistema distribuido (dirección IP, puerto de escucha e identificador).
- El contenido de la referencia remota no es directamente accesible, sino que es gestionado por el stub y por el enlazador.
- **Enlazador**: servicio de un sistema distribuido que registra las asignaciones de nombres a referencias remotas. Mantiene una tabla con pares *(nombre, referencia remota)*.
  - En el cliente, se usa para obtener la referencia de un objeto remoto a partir de su nombre (`lookup()`).
  - En el servidor, se usa para registrar o ligar (`bind()`) sus objetos remotos por nombre de forma que los clientes pueden buscarlos.

Java RMI

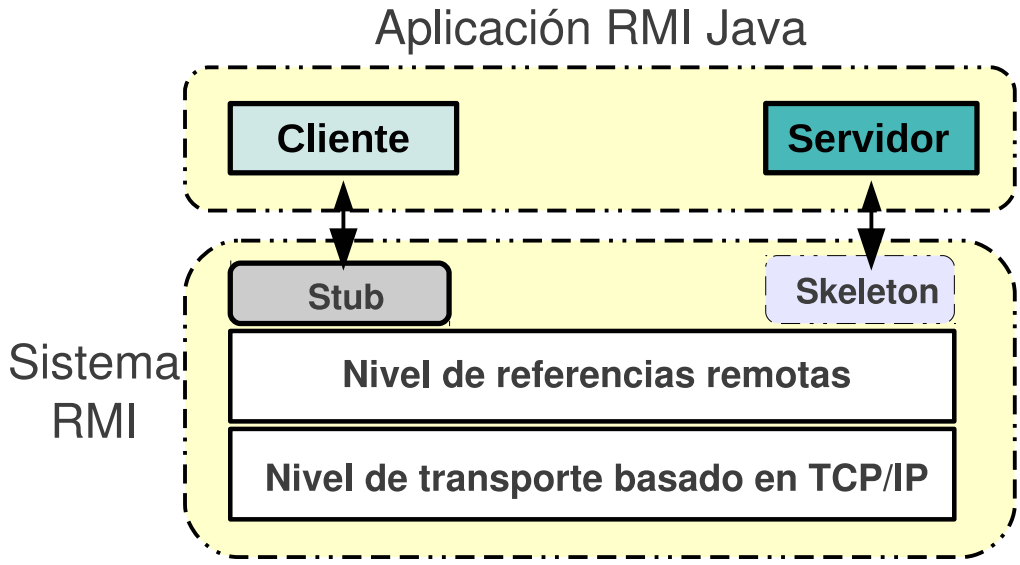
- Mecanismo que ofrece Java (desde JDK 1.1) para invocar métodos de objetos remotos en diferentes JVMs.
- Permite que un programa Java exporte un objeto para que esté disponible en la red, esperando conexiones desde objetos ejecutándose en otras máquinas virtuales Java (JVM).
- La idea es permitir la programación de aplicaciones distribuidas de forma integrada en Java como si se tratara de aplicaciones locales, preservando la mayor parte de la semántica de objetos en Java.
- Permite mantener seguro el entorno de la plataforma Java mediante gestores de seguridad y cargadores de clases.

Particularidades de Java RMI

- Las **interfaces remotas** se definen como cualquier interfaz Java pero deben extender una interfaz denominada `Remote` y lanzar excepciones remotas para actuar como tales.
- En una llamada remota, los parámetros de un método son todos de entrada y la salida es el resultado de la llamada.
  - Los parámetros que son objetos remotos se pasan por referencia.
  - Los parámetros que son objetos locales (argumentos no remotos) y los resultados se pasan por valor (las referencias solo tienen sentido dentro de la misma JVM).
  - Cuando el servidor no tiene la implementación de un objeto local, la JVM se encarga de descargar la clase asociada a dicho objeto.
- En Java RMI, el enlazador se denomina `rmiregistry`.

Arquitectura de Java RMI

- Generalmente las aplicaciones RMI constan de **servidor** y **clientes**.
- El sistema RMI proporciona los mecanismos para que el servidor y los clientes se comuniquen e intercambien información.



Arquitectura de Java RMI. Nivel de aplicación

Nivel de Aplicación: Los clientes y el servidor implementan la funcionalidad de la aplicación RMI:

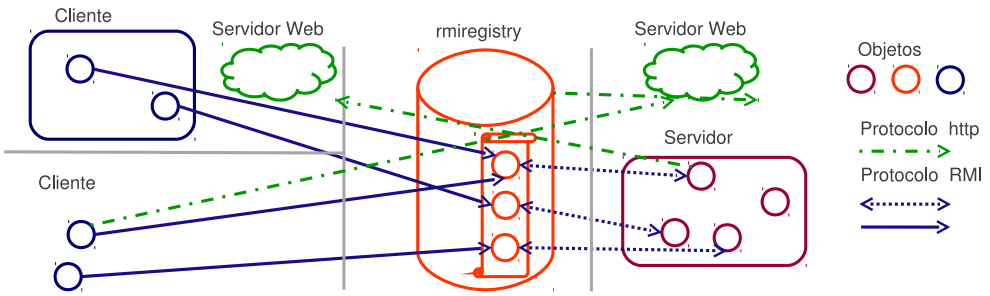
- **Servidor**:
  - Crea objetos remotos.
  - Hace accesible las referencias a dichos objetos remotos. Para ello, se registran los objetos remotos en el `rmiregistry`.
  - Se mantiene activo esperando la invocación de mñetodos sobre dichos objetos remotos por parte de los clientes.
- **Clientes**:
  - Deben declarar objetos de la interfaz remota.
  - Deben obtener referencias a objetos remotos (stubs) en el servidor.
  - Invocan los métodos remotos (usando el stub).

Este tipo de aplicaciones se suelen denominar **Aplicaciones de objetos distribuidos**

Arquitectura de Java RMI. Nivel de aplicación(2)

Las aplicaciones de objetos distribuidos requieren:

- **Localizar objetos remotos**:
  - Para ello, las aplicaciones pueden registrar sus objetos remotos utilizando el servicio `rmiregistry`, o puede enviar y devolver referencias a objetos remotos como argumentos y resultados.
- **Comunicarse con objetos remotos**:
- **Cargar bytecodes de objetos** que se pasan como parámetros o valores de retorno.



Arquitectura de Java RMI. Sistema RMI

- **Nivel de stubs**
  - El papel del `Skeleton` lo hace la plataforma RMI (a partir de JDK 1.2).
  - El stub se genera automáticamente bajo demanda en tiempo de ejecución (a partir de JDK 1.5, antes se usaba un compilador).
  - El stub tiene la misma interfaz que el objeto remoto y conoce su localización. Realiza todo el *marshalling* necesario para la llamada remota.
  - Debe haber un stub por cada instancia de una interfaz remota.
- **Nivel de referencias**: Se encarga de:
  - Interpretar las referencias remotas que manejan los stubs para permitirles acceder a los métodos correspondientes de los objetos remotos
  - Enviar y recibir los paquetes (resultantes del *marshalling* de las peticiones/respuestas) usando los servicios del nivel de transporte.
- **Nivel de transporte**: Se encarga de conectar las diferentes JVMs en el sistema distribuido. Se basa en el protocolo de red TCP/IP.

Pasos para implementar una aplicación Java RMI

Suponiendo, por simplicidad, que el cliente usa un único objeto remoto, los pasos serían:

1. **Crear la interfaz remota**
2. **Implementar el objeto remoto**
3. **Implementar los programas clientes**

1. Crear la Interfaz remota

- La interfaz remota declara los métodos que pueden invocar remotamente los clientes.
- Deben extender la interfaz `Remote` del paquete `java.rmi`.
- Los métodos definidos deben poder lanzar una excepción remota.

Ejemplo: Interfaz de un contador remoto `Contador_I.java`

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Contador_I extends Remote {
 public void incrementa(int valor) throws RemoteException;
 public int getvalor() throws RemoteException;
}
```

2. Implementar el objeto remoto

- Inicialmente se deben de definir los métodos de la interfaz remota:

Ejemplo: Clase para contador remoto `Contador.java`

```
import java.rmi.*;
import java.rmi.server.*;

public class Contador implements Contador_I{
 private int valor;

 public Contador() throws RemoteException {
     valor=0;
 }

 public void incrementa(int valor) throws RemoteException {
      this.valor+=valor;
 }

 public int getvalor() throws RemoteException {
      return valor;
 }
...}
```

2. Implementar el objeto remoto(2)

Una vez definidos los métodos de la interfaz, se debe crear el objeto remoto que implementa la interfaz remota y exportarlo al entorno RMI para habilitar la recepción de invocaciones remotas. Eso implica:

1. Crear e instalar un **gestor de seguridad** (*security manager*).
2. Crear y exportar el objeto remoto.
3. Se ha de registrar la referencia a dicho objeto remoto (el stub) en el servidor de nombre (`rmiregistry`) asociándole un nombre. Esto se hace con `rebind(nombre, referencia)`.

Estos pasos se podrían incluir en el programa ppal de la clase que implementa la interfaz remota (como en el siguiente ejemplo) o en cualquier otro método de otra clase.

2. Implementar el objeto remoto(3)

Ejemplo: método `main` para contador remoto `Contador.java`

```
...

public static void main(String[] args) {
 // Instalacion del gestor de seguridad
 if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
 }
 try {
    String nombre = "contador";
    Creacion de una instancia de la clase
    Contador_I cont = new Contador();
    La exportamos y le damos nombre en el RMI registry
    Contador_I stub =(Contador_I)
          UnicastRemoteObject.exportObject(cont, 0);
    Registry registry = LocateRegistry.getRegistry();
    registry.rebind(nombre,stub);
    System.out.println("Contador ligado");
   } catch (Exception e) {
       System.err.println("Contador exception:");
       e.printStackTrace();
     }
 }
```

3. Implementar los programas clientes

Implementar los programas clientes

- El cliente también debe instalar un gestor de seguridad para que el stub local pueda descargar la definición de una clase desde el servidor.
- El programa cliente deberá obtener la referencia del objeto remoto, consultando el servicio de enlazador, `rmiregistry`, en la máquina del servidor, usando `lookup(...)`.

- Una vez obtenida la referencia remota, el programa interactúa con el objeto remoto invocando métodos del stub usando los argumentos adecuados (como si fuera local).

3. Implementar los programas clientes (2)

Ejemplo: Programa cliente del contador remoto `Cliente.java`

```
import java.rmi.*;

public class Cliente {

 public static void main(String args[]) {
  if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
  }
  try {
   // Obteniendo una referencia a RMI registry en nodo servidor
   // El primer argumento del programa debe ser el nodo servidor
   Registry registry = LocateRegistry.getRegistry(args[0]);
   //Se invoca lookup en el registry para buscar objeto remoto
   // mediante el nombre usado en la clase Contador
   Contador_I cont = (Contador_I)(registry.lookup("contador");
   cont.incrementa(2);
   System.out.println("VALOR="+cont.getvalor());
  } catch (Exception e) {
            System.err.println("Contador_I exception:");
            e.printStackTrace();
    }
 }
}
```

Bibliografía del tema 3.

Para más información, ejercicios, bibliografía adicional, se puede consultar:

3.1. **Mecanismos básicos en sistemas basados en paso de mensajes.** Palma (2003), capítulos 7,8,9. Almeida (2008), capítulo 3. Kumar (2003), capítulo 6.

3.2. **Paradigmas de interacción de procesos en programas distribuidos** Andrews (2000), capítulo 9. Almeida (2008), capítulos 5,6.

3.3. **Mecanismos de alto nivel en sistemas distribuidos. RPC y RMI.** Palma (2003), capítulo 10. Coulouris (2011), capítulo 5.

4.4 Problemas del tema 3.

28

En un sistema distribuido, 6 procesos clientes necesitan sincronizarse de forma específica para realizar cierta tarea, de forma que dicha tarea sólo podrá ser realizada cuando tres procesos estén preparados para realizarla. Para ello, envían peticiones a un proceso controlador del recurso y esperan respuesta para poder realizar la tarea específica. El proceso controlador se encarga de asegurar la sincronización adecuada. Para ello, recibe y cuenta las peticiones que le llegan de los procesos, las dos primeras no son respondidas y producen la suspensión del proceso que envía la petición (debido a que se bloquea esperando respuesta) pero la tercera petición produce el desbloqueo de los tres procesos pendientes de respuesta. A continuación, una vez desbloqueados los tres procesos que han pedido (al recibir respuesta), inicializa la cuenta y procede cíclicamente de la misma forma sobre otras peticiones.

El código de los procesos clientes es el siguiente, asumiendo que se usan operaciones síncronas.

```
process Cliente[ i : 0..5 ] ;
begin
   while true do begin
      send( peticion, Controlador );
      receive( permiso, Controlador );
      Realiza_tarea_grupal( );
   end
end
```

```
process Controlador ;
begin
   while true do begin

      ...

   end
end
```

Describir en pseudocódigo el comportamiento del proceso controlador, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos. Es posible utilizar una sentencia del tipo `select for i=... to ...` para especificar diferentes ramas de una sentencia selectiva que comparten el mismo código dependiente del valor de un índice `i`.

29

En un sistema distribuido, 3 procesos productores producen continuamente valores enteros y los envían a un proceso buffer que los almacena temporalmente en un array local de 4 celdas enteras para ir enviándoselos a un proceso consumidor. A su vez, el proceso buffer realiza lo siguiente, sirviendo de forma equitativa al resto de procesos:

a) Envía enteros al proceso consumidor siempre que su array local tenga al menos dos elementos disponibles.

b) Acepta envíos de los productores mientras el array no esté lleno, pero no acepta que cualquier productor pueda escribir dos veces consecutivas en el búfer.

El código de los procesos productor y consumidor es el siguiente, asumiendo que se usan operaciones síncronas.

```
process Productor[ i : 0..2 ] ;
begin
   while true do begin
      Produce(&dato );
      send( &dato, Buffer );
   end
end
```

```
process Consumidor ;
begin
   while true do begin
      receive ( &dato, Buffer );
      Consume (dato);
   end
end
```

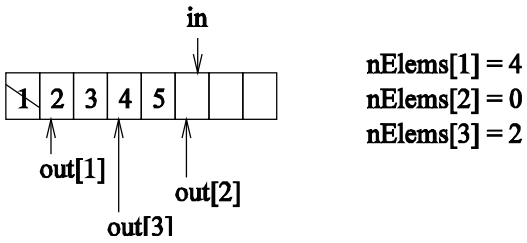
Describir en pseudocódigo el comportamiento del proceso Buffer, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos.

```
process Buffer ;
begin
   while true do begin
      ...
   end
end
```

30

Suponer un proceso productor y 3 procesos consumidores que comparten un buffer acotado de tamaño B. Cada elemento depositado por el proceso productor debe ser retirado por todos los 3 procesos consumidores para ser eliminado del buffer. Cada consumidor retirará los datos del buffer en el mismo orden en el que son depositados, aunque los diferentes consumidores pueden ir retirando los elementos a ritmo diferente unos de otros. Por ejemplo, mientras un consumidor ha retirado los elementos 1, 2 y 3, otro consumidor puede haber retirado solamente el elemento 1. De esta forma, el consumidor más rápido podría retirar hasta B elementos más que el consumidor más lento.

Describir en pseudocódigo el comportamiento de un proceso que implemente el buffer de acuerdo con el esquema de interacción descrito usando una construcción de espera selectiva, así como el del proceso productor y de los procesos consumidores. Comenzar identificando qué información es necesario representar, para después resolver las cuestiones de sincronización. Una posible implementación del bufffer mantendría, para cada proceso consumidor, el puntero de salida y el número de elementos que quedan en el buffer por consumir (ver figura).



31

Una tribu de antropófagos comparte una olla en la que caben M misioneros. Cuando algún salvaje quiere

comer, se sirve directamente de la olla, a no ser que ésta esté vacía. Si la olla está vacía, el salvaje despertará al cocinero y esperará a que éste haya rellenado la olla con otros M misioneros.

```
process Salvaje[ i : 0..2 ] ;
begin
   while true do begin
      { esperar a servirse un misionero: }
      .....
      { comer }
      Comer();
   end
end
```

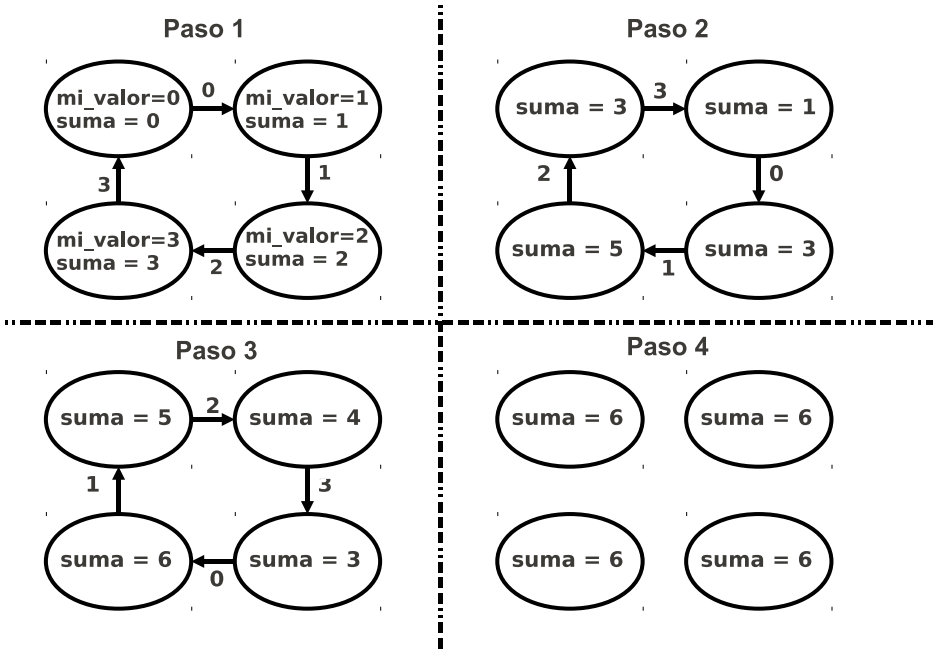
```
process Cocinero ;
begin
   while true do begin
      { dormir esperando solicitud para llenar: }
      ......
      { rellenar olla: }
      ......
   end
end
```

Implementar los procesos salvajes y cocinero usando paso de mensajes, usando un proceso olla que incluye una construcción de espera selectiva que sirve peticiones de los salvajes y el cocinero para mantener la sincronización requerida, teniendo en cuenta que:

- La solución no debe producir interbloqueo.
- Los salvajes podrán comer siempre que haya comida en la olla,
- Solamente se despertará al cocinero cuando la olla esté vacía.
- Los procesos usan operaciones de comunicación síncronas.

32

Considerar un conjunto de N procesos, `P[i]`, $(i = 0, ..., N-1)$ conectados en forma de anillo. Cada proceso tiene un valor local almacenado en su variable local `mi_valor`. Deseamos calcular la suma de los valores locales almacenados por los procesos de acuerdo con el algoritmo que se expone a continuación.



Los procesos realizan una serie de iteraciones para hacer circular sus valores locales por el anillo. En la primera iteración, cada proceso envía su valor local al siguiente proceso del anillo, al mismo tiempo que recibe del proceso anterior el valor local de éste. A continuación acumula la suma de su valor local y el recibido desde el proceso anterior. En las siguientes iteraciones, cada proceso envía al siguiente proceso siguiente el valor recibido en la anterior iteración, al mismo tiempo que recibe del proceso anterior un nuevo valor. Después acumula la suma. Tras un total de $N-1$ iteraciones, cada proceso conocerá la suma de todos los valores locales de los procesos.

Dar una descripción en pseudocódigo de los procesos siguiendo un estilo SPMD y usando operaciones de envío y recepción síncronas.

```
process P[ i : 0..N-1 ] ;
   var mi_valor : integer := ... ;
       suma     : integer ;
begin
   for j := 0 to N-1 do begin
      ...
   end
end
```

33

Considerar un estanco en el que hay tres fumadores y un estanquero. Cada fumador continuamente lía un cigarro y se lo fuma. Para liar un cigarro, el fumador necesita tres ingredientes: tabaco, papel y cerillas. Uno de los fumadores tiene solamente papel, otro tiene solamente tabaco, y el otro tiene solamente cerillas. El estanquero tiene una cantidad infinita de los tres ingredientes.

- El estanquero coloca aleatoriamente dos ingredientes diferentes de los tres que se necesitan para

hacer un cigarro, desbloquea al fumador que tiene el tercer ingrediente y después se bloquea. El fumador seleccionado, se puede obtener fácilmente mediante una función **genera_ingredientes** que devuelve el índice (0,1, ó 2) del fumador escogido.

- El fumador desbloqueado toma los dos ingredientes del mostrador, desbloqueando al estanquero, lía un cigarro y fuma durante un tiempo.
- El estanquero, una vez desbloqueado, vuelve a poner dos ingredientes aleatorios en el mostrador, y se repite el ciclo.

Describir una solución distribuida que use envío asíncrono seguro y recepción síncrona, para este problema usando un proceso **Estanquero** y tres procesos fumadores **Fumador**`(i)` (con `i`=0,1 y 2).

```
process Estanquero ;
begin
   while true do begin
      ....
   end
end
```

```
process Fumador[ i : 0..2 ] ;
begin
   while true do begin
      ....
   end
end
```

34

En un sistema distribuido, un gran número de procesos clientes usa frecuentemente un determinado recurso y se desea que puedan usarlo simultáneamente el máximo número de procesos. Para ello, los clientes envían peticiones a un proceso controlador para usar el recurso y esperan respuesta para poder usarlo (véase el código de los procesos clientes). Cuando un cliente termina de usar el recurso, envía una solicitud para dejar de usarlo y espera respuesta del Controlador. El proceso controlador se encarga de asegurar la sincronización adecuada imponiendo una única restricción por razones supersticiosas: nunca habrá 13 procesos exactamente usando el recurso al mismo tiempo.

```
process Cli[ i : 0....n ] ;
var pet_usar    : integer := 1 ;
    pet_liberar : integer := 2 ;
    permiso     : integer := ... ;
begin
   while true do begin
      send( pet_usar, Controlador );
      receive( permiso, Controlador );

      Usar_recurso( );

      send( pet_liberar, Controlador );
      receive( permiso, Controlador );
   end
end
```

```
process Controlador ;
begin
   while true do begin
     select

       ...

   end
end
```

Describir en pseudocódigo el comportamiento del proceso controlador, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos. Es posible utilizar una sentencia del tipo `select for i=... to ...` para especificar diferentes ramas de una sentencia selectiva que comparten el mismo código dependiente del valor de un índice `i`.

35

En un sistema distribuido, tres procesos **Productor** se comunican con un proceso **Impresor** que se encarga de ir imprimiendo en pantalla una cadena con los datos generados por los procesos productores. Cada proceso productor (**Productor**`[`i`]` con $i = 0,1,2$) genera continuamente el correspondiente entero i, y lo envía al proceso **Impresor**.

El proceso **Impresor** se encarga de ir recibiendo los datos generados por los productores y los imprime por pantalla (usando el procedimiento `imprime(`*entero*`)`) generando una cadena dígitos en la salida. No obstante, los procesos se han de sincronizar adecuadamente para que la impresión por pantalla cumpla las siguientes restricciones:

- Los dígitos 0 y 1 deben aceptarse por el impresor de forma alterna. Es decir, si se acepta un 0 no podrá volver a aceptarse un 0 hasta que se haya aceptado un 1, y viceversa, si se acepta un 1 no podrá volver a aceptarse un 1 hasta que se haya aceptado un 0.
- El número total de dígitos 0 o 1 aceptados en un instante no puede superar el doble de número de digitos 2 ya aceptados en dicho instante.

Cuando un productor envía un digito que no se puede aceptar por el imprersor, el productor quedará bloqueado esperando completar el **send**.

El pseudocódigo de los procesos productores (**Productor**) se muestra a continuación , asumiendo que se usan operaciones bloqueantes no buferizadas (síncronas).

```
process Productor[ i : 0,1,2 ]
while true do begin
   send( i, Impresor ) ;
end
```

Escribir en pseudocódigo el código del proceso **Impresor**, utilizando un bucle infinito con una orden de espera selectiva **select** que permita implementar la sincronización requerida entre los procesos, según este esquema:

```
Process Impresor
var
  .....
begin
  while true do begin
     select
        .....
     end
  end
end
```

36

En un sistema distribuido hay un vector de `n` procesos iguales que envían con **`send`** (en un bucle infinito) valores enteros a un proceso receptor, que los imprime.

Si en algún momento no hay ningún mensaje pendiente de recibir en el receptor, este proceso debe de imprimir "no hay mensajes. duermo." y después bloquearse durante 10 segundos (con **`sleep`**`(10)`), antes de volver a comprobar si hay mensajes (esto podría hacerse para ahorrar energía, ya que el procesamiento de mensajes se hace en ráfagas separadas por 10 segundos).

Este problema no se puede solucionar usando **`receive`** o **`i_receive`**. Indica a que se debe esto. Sin embargo, sí se puede hacer con **`select`**. Diseña una solución a este problema con **`select`**.

```
process Emisor[ i : 1..n ]
    var dato : integer ;
begin
   while true do begin
        dato := Producir() ;
        send( dato, Receptor );
   end
end
process Receptor()
   var dato : integer ;
begin
   while true do
     ......
end
```

37

En un sistema tenemos N procesos emisores que envían de forma segura un único mensaje cada uno de ellos a un proceso receptor, mensaje que contiene un entero con el número de proceso emisor. El proceso receptor debe de imprimir el número del proceso emisor que inició el envío en primer lugar. Dicho emisor debe terminar, y el resto quedarse bloqueados.

```
process Emisor[ i : 1.. N ]
begin
   s_send(i,Receptor);
end
process Receptor ;
   var ganador : integer ;
begin
   { calcular 'ganador' }
   ....
   ....
   print "El primer envio lo ha realizado: ....", ganador ;
end
```

Para cada uno de los siguientes casos, describir razonadamente si es posible diseñar una solución a este problema o no lo es. En caso afirmativo, escribe una posible solución:

(a) el proceso receptor usa exclusivamente recepción mediante una o varias llamadas a **receive**

(b) el proceso receptor usa exclusivamente recepción mediante una o varias llamadas a **i_receive**

(c) el proceso receptor usa exclusivamente recepcíon mediante una o varias instrucciones **select**

38

Supongamos que tenemos N procesos concurrentes semejantes:

```
process P[ i : 1..N ] ;
   ....
begin
   ....
end
```

Cada proceso produce N-1 caracteres (con N-1 llamadas a la función **ProduceCaracter**) y envía cada carácter a los otros N-1 procesos. Además, cada proceso debe imprimir todos los caracteres recibidos de los otros procesos (el orden en el que se escriben es indiferente).

(a) Describe razonadamente si es o no posible hacer esto usando exclusivamente **s_send** para los envíos. En caso afirmativo, escribe una solución.

(b) Escribe una solución usando **send** y **receive**

39

Escribe una nueva solución al problema anterior en la cual se garantize que el orden en el que se imprimen los caracteres es el mismo orden en el que se inician los envíos de dichos caracteres (pista: usa **select** para recibir).

40

Supongamos de nuevo el problema anterior en el cual todos los procesos envían a todos. Ahora cada item de datos a producir y transmitir es un bloque de bytes con muchos valores (por ejemplo, es una imagen que puede tener varios megabytes de tamaño). Se dispone del tipo de datos **Tipo_bloque** para ello, y el procedimiento **ProducirBloque**, de forma que si `b` es una variable de tipo **Tipo_bloque**, entonces la llamada a **ProducirBloque**`(b)` produce y escribe una secuencia de bytes en `b`. En lugar de imprimir los datos, se deben consumir con una llamada a **ConsumirBloque**`(b)`.

Cada proceso se ejecuta en un ordenador, y se garantiza que hay la suficiente memoria en ese ordenador como para contener simultáneamente al menos hasta N bloques. Sin embargo, el sistema de paso de mensajes

(SPM) podría no tener memoria suficiente como para contener los $(N-1)^2$ mensajes en tránsito simultáneos que podría llegar a haber en un momento dado con la solución anterior.

En estas condiciones, si el SPM agota la memoria, debe retrasar los **`send`** dejando bloqueados los procesos y en esas circunstancias se podría producir interbloqueo. Para evitarlo, se pueden usar operaciones inseguras de envío, **`i_send`**. Escribe dicha solución, usando como orden de recepción el mismo que en el problema anterior (3).

41

En los tres problemas anteriores, cada proceso va esperando a recibir un item de datos de cada uno de los otros procesos, consume dicho item, y después pasa recibir del siguiente emisor (en distintos órdenes). Esto implica que un envío ya iniciado, pero pendiente, no puede completarse hasta que el receptor no haya consumido los anteriores bloques, es decir, se podría estar consumiendo mucha memoria en el SPM por mensajes en tránsito pendientes cuya recepción se ve retrasada.

Escribe una solución en la cual cada proceso inicia sus envíos y recepciones y después espera a que se completen todas las recepciones antes de iniciar el primer consumo de un bloque recibido. De esta forma todos los mensajes pueden transferirse potencialmente de forma simultánea. Se debe intentar que la transimisión y las producción de bloques sean lo más simultáneas posible. Suponer que cada proceso puede almacenar como mínimo 2N bloques en su memoria local, y que el orden de recepción o de consumo de los bloques es indiferente.

Chapter 5

Tema 4. Introducción a los sistemas de tiempo real.

5.1 Concepto de sistema de tiempo real. Medidas de tiempo y modelo de tareas.

5.1.1 Definición, tipos y ejemplos

Sistemas de Tiempo Real

- Constituyen un tipo de sistema en el que la ejecución del sistema se debe producir dentro de unos plazos de tiempo predefinidos para que funcione con la suficiente garantía.
- En un sistema además concurrente será necesario que todos los procesos sobre un procesador o sobre varios se ejecuten en los plazos de tiempo predefinidos.

Definición de un Sistema de Tiempo Real

Stankovic (1997) da la siguiente definición:

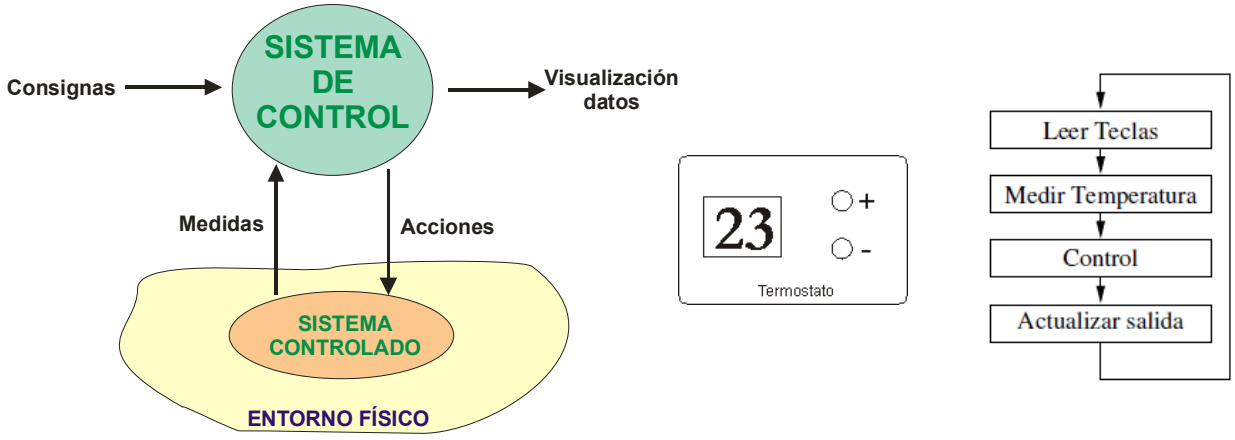
> Un *sistema de tiempo real* es aquel sistema cuyo funcionamiento correcto depende no sólo de resultados lógicos producidos por el mismo, sino también del instante de tiempo en el que se producen esos resultados

Corrección Funcional + Corrección Temporal

> El no cumplimiento de una restricción temporal lleva a un **fallo** del sistema

Ejemplo: Sistema de Control

- Objetivo: Ejecutar el lazo de control del sistema en instantes de tiempo prefijados.
- Condición de tiempo real: no puede haber retrasos en la ejecución del lazo de control, ya que afecta al rendimiento y provoca perdida de estabilidad.



Los sistemas de control (no necesariamente computadores sino sistemas eléctricos, mecánicos, ópticos) suelen realizar la acción de corrección para el control del sistema en un lazo (bucle) con el objeto de mantener o establecer una condiciones en el sistema controlado. En la actualidad los sistemas de control se están "digitalizando", es decir, sustituyendo por sistemas de control computerizados.

La estructura general de una aplicación de control incluye:

- Sistema de control: que se encarga de monitorizar el sistema controlado, y si se produce una desviación respecto del comportamiento esperado se activan las señales de control (acciones) especificas para disminuir esa desviación.
- Sensores: dispositivos de naturaleza electrómagnética, mecánica u óptica que obtienen el estado del entorno físico a partir de la cuantificación de alguna variable física. Los sensores se encargan de realizar la transducción entre valores en alguna variable física y la valores de tensión (los que se utilizan desde cualquier medio computerizado).

  Ejemplos: sensores de temperatura, sensores de presencia, de luminosidad, tacómetros, ...
- Actuadores: dispositivos de naturaleza electromagnética, mecánica u óptica que realizan acciones concretas en el entorno físico a partir de las señales de control enviadas por el sistema de control. La traducción pasa valores de tensión en acciones concretas.

  Ejemplos: motores, lamparas, valvulas, ...
- Sistema Controlado: es el sistema físico del que queremos controlar su comportamiento.
- Consignas: son las ordenes que realizan los usuarios para establecer el sistema controlado en un valor de referencia.

- Visualización de datos: se obtienen los datos para que el operador pueda visualizarlos.

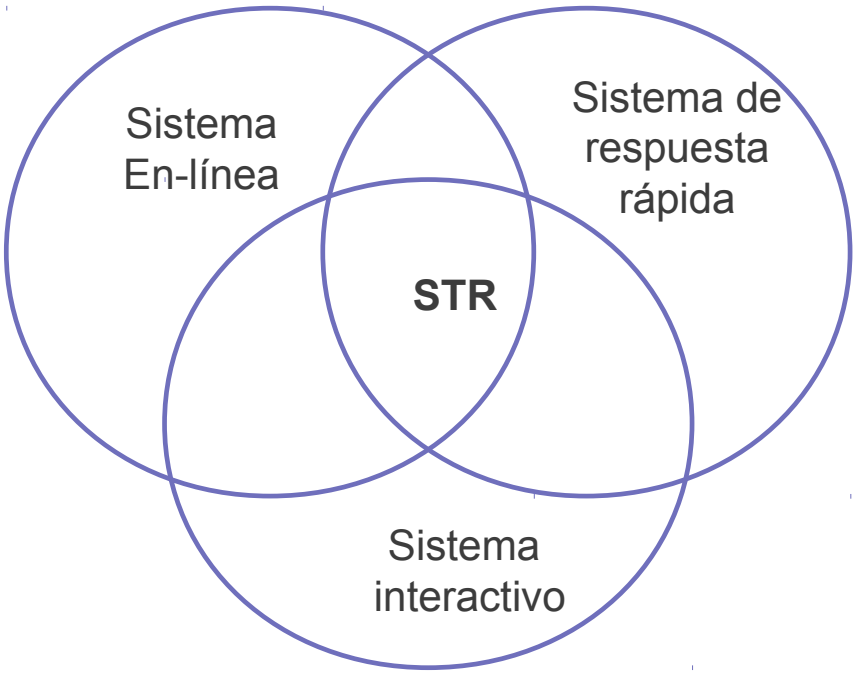
¿Qué tipo de sistema de tiempo real es un sistema de control? Depende del tipo de actividad, pero para actividades de control debería ser de tipo estricto.

Tipos de Sistemas de Tiempo Real

Habitualmente suele asociarse la denominación de sistemas de tiempo real a los siguientes tipos de sistemas:

- **Sistema "en-línea":** Siempre está disponible, pero no se garantiza una respuesta en intervalo de tiempo acotado.
  - Ejemplos: Cajeros automáticos, sistemas de reservas de vuelo.
- **Sistema interactivo:** El sistema ofrece una respuesta en un tiempo, aunque no importa el tiempo que necesita para su ejecución.
  - Ejemplos: Reproductor DVD, sistema aire acondicionado, juegos, ..
- **Sistema de respuesta rápida:** El sistema ofrece una respuesta en el menor tiempo posible y de forma inmediata.
  - Ejemplos: Sistema anti incendios, alarmas, etc.

Tipos de Sistemas de Tiempo Real



Ejemplos de Sistemas de Tiempo Real

En la actualidad hay muchos ejemplos de uso de Sistemas de Tiempo Real, podemos citar algunos de ellos:

- **Automoción:** sistema de ignición, transmisión, dirección asistida, frenos antibloqueo (ABS), control de la tracción, ...
- **Electrodomésticos:** televisores, lavadoras, hornos de microondas, reproductores de videos o DVDs, sistemas de seguridad y vigilancia, ...
- **Aplicaciones aeroespaciales:** control de vuelos, controladores de motores, pilotos automáticos, ...
- **Equipamiento médico:** como sistemas de monitorización de anestesia, monitores ECG,
- **Sistemas de defensa:** como sistemas radar de aviones, sistemas de radio, control de misiles, ...

5.1.2 Propiedades de los Sistemas de Tiempo Real

Propiedades de un STR

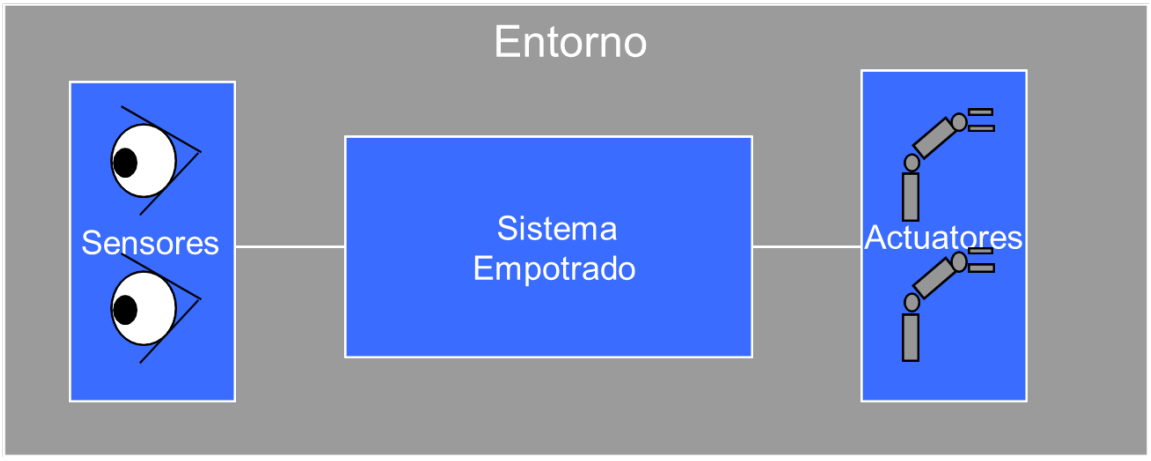
Al depender la corrección del sistema de las restricciones temporales, los sistemas de tiempo real tienen que cumplir una serie de propiedades:

- **Reactividad.**
- **Predecibilidad.**
- **Confiabilidad.**

a continuación veremos cada una de estas propiedades con más detalle.

Propiedad: Reactividad

El sistema tiene que interaccionar con el entorno, y responder de la manera esperada a los estímulos externos dentro de un intervalo de tiempo previamente definido.



Propiedad: Predecibilidad

Tener un comportamiento predecible implica que la ejecución del sistema tiene que ser determinista, y por lo tanto, se debe garantizar su ejecución dentro del plazo de tiempo definido.

- Las respuestas han de producirse dentro de las restricciones temporales impuestas por el entorno (sistema controlado), y que suele ser diferente para cada proceso del sistema.
- Es necesario conocer el comportamiento temporal de los componentes software (SO, middleware, librería, etc) y hardware utilizados, así como del lenguaje de programación.
- Si no se puede tener un conocimiento temporal exacto, hay que definir marcos de tiempo acotados; p.e. conocer el tiempo de peor ejecución de un algoritmo, el tiempo máximo de acceso a un dato de E/S)

Propiedad: Confiabilidad

- La **Confiabilidad** (***Dependability***) mide el grado de confianza que se tiene del sistema. Depende de:
  - Disponibilidad (*avalaibility*). Capacidad de proporcionar servicios siempre que se solicita.
  - Robustez o tolerancia a fallos (*fault tolerant*). Capacidad de operar en situaciones excepcionales sin poseer un comportamiento catastrófico.
  - Fiabilidad (*reliability*). Capacidad de ofrecer siempre los mismos resultados bajo las mismas condiciones.
  - Seguridad: Capacidad de protegerse ante ataques o fallos accidentales o deliberados (*safety*), y a la no vulnerabilidad de los datos (*security*).
- Cuando esta propiedad es crítica (su no cumplimiento lleva a pérdida humana o económica), el sistema se denomina sistema de tiempo real **crítico** (***safety-critical system***)
  - Ejemplos: Sistemas de Aviónica, Sistemas de automoción, Centrales nucleares, etc.

Tipos de STRs (otra clasif. distinta)

Tipo	Características	Ejemplos
No permisivos o estrictos (*hard*)	Plazo de respuesta dentro del límite preestablecido.	Control de vuelo de una aeronave. Sistema automático de frenado ABS.
Permisivos, flexibles o no estrictos (*soft*)	Aunque el plazo de respuesta es importante, el sistema funciona correctamente aunque no respondan en el plazo de tiempo fijado	Sistema de adquisición de datos meteorológicos.
Firmes	Se permiten algunos fallos en el plazo de respuesta, pero un número excesivo provoca una fallo completo del sistema	Sistema de control de reserva de vuelos. Sistema de video bajo demanda.

Los sistemas de tiempo real pueden tener componentes de las tres clases.

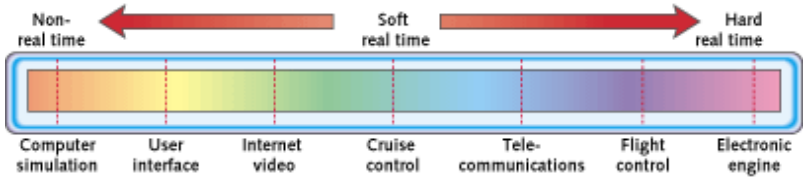


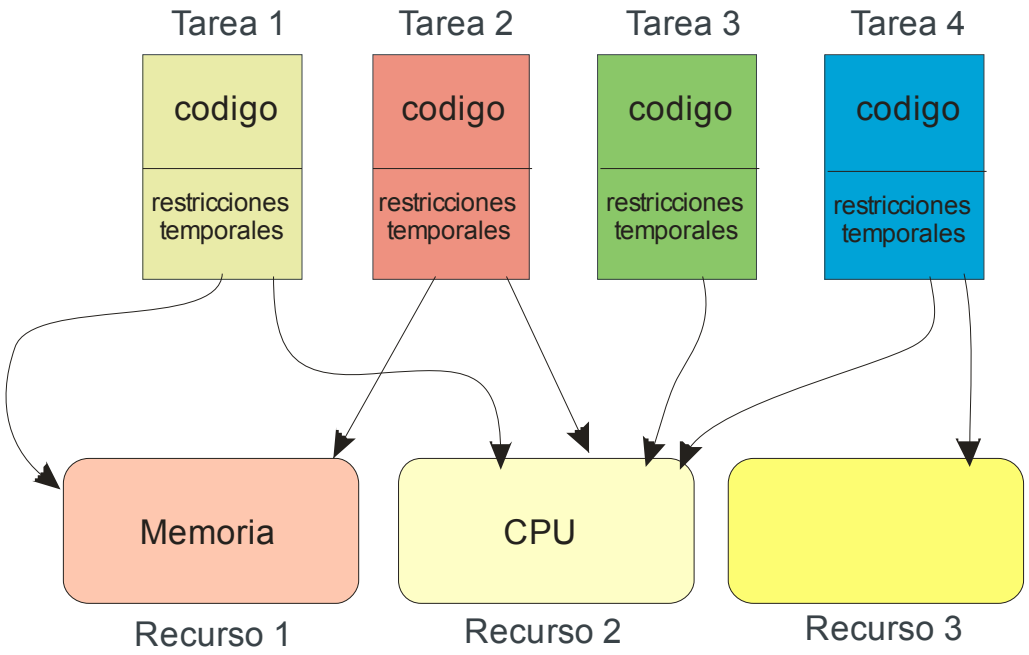
Imagen obtenida de:

`http://www.embedded.com/electronics-blogs/beginner-s-corner/4023859/Introduction-to-Real-Time`

5.1.3 Modelo de Tareas

Modelo de Tareas

- Un sistema de tiempo real se estructura en **tareas** que acceden a los recursos del sistema.



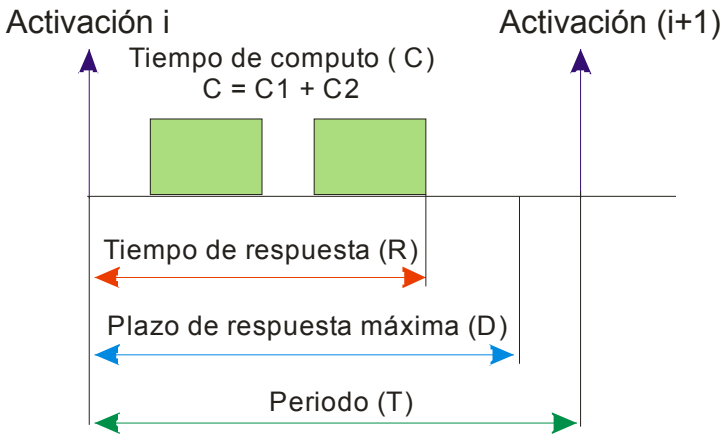
Tareas y recursos. Definición.

En los STR cabe distinguir estos dos tipos de elementos:

- **Tarea:** El conjunto de acciones que describe el comportamiento del sistema o parte de él en base a la ejecución secuencial de una sección de código (o programa). Equivalente al *proceso* o *hebra*.
  - La tarea satisface una necesidad funcional concreta.
  - La tarea tiene definida unas restricciones temporales a partir de los **Atributos Temporales**.
- **Recursos:** Elementos disponibles para la ejecución de las tareas.
  - Recursos Activos: Procesador, Red, ...
  - Recursos Pasivos: Datos, Memoria, Dispositivos de E/S, ...

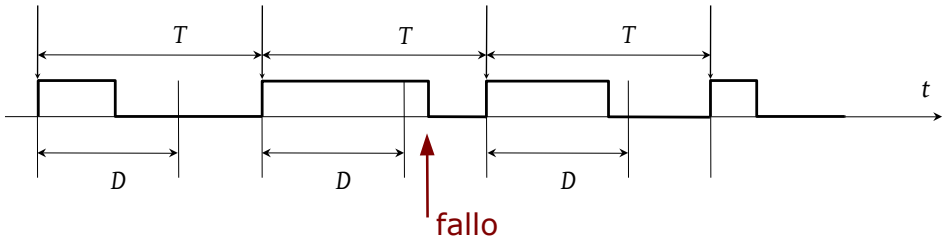
Atributos temporales

- Tiempo de computo o de ejecución (C): Tiempo necesario para la ejecución de la tarea.
- Tiempo de respuesta (R): Tiempo que ha necesitado el proceso para completarse totalmente.
- Plazo de respuesta máxima (*deadline*) (D): Define el máximo tiempo de respuesta posible.
- Periodo (T): Intervalo de tiempo entre dos activaciones sucesivas en el caso de una tarea periódica.

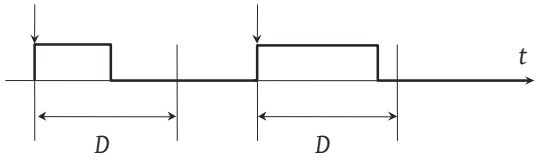


Tipos de Tareas

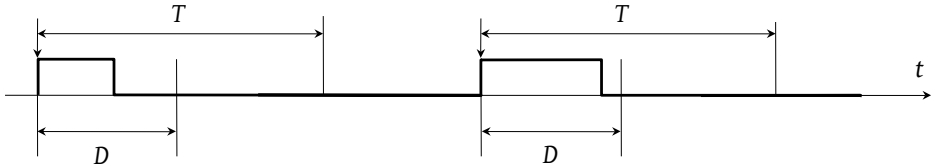
- **Periódica:** T es el período de activación de la tarea.



- **Aperiódica:**

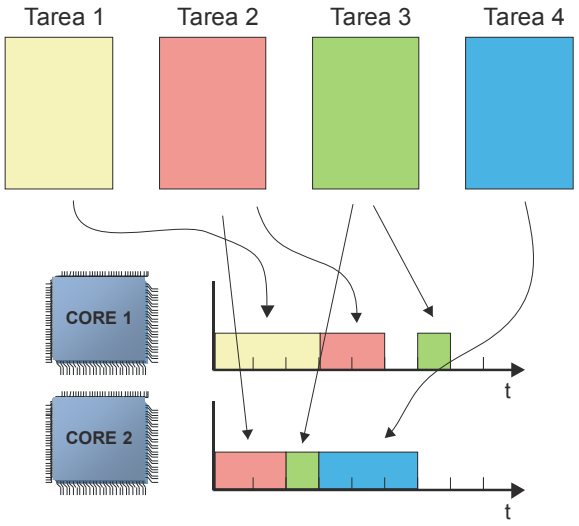


- **Esporádica:** T es la separación mínima entre eventos.



Planificación de Tareas

- La **planificación** de tareas consiste en asignar las tareas a los recursos activos de un sistema (principalmente el procesador o procesadores) para garantizar la ejecución de todas las tareas de acuerdo a un conjunto de restricciones especificas.
  - En tareas de tiempo real las restricciones suelen estar asociadas a restricciones temporales como los plazos límites.



Diseño de la planificación de tareas.

El problema de la planificación supone:

- Determinar los procesadores disponibles a los que se puede asociar las tareas.
- Determinar las relaciones de dependencias de las tareas:
  - Relaciones de precedencia que hay entre las distintas tareas.
  - Determinar los recursos comunes a los que acceden las distintas tareas.
- Determinar el orden de ejecución de la tareas para garantizar las restricciones especificadas.

Esquema de planificación de tareas

Para determinar la planificabilidad de un conjunto de tareas se requiere un esquema de planificación que cubra los dos siguientes aspectos:

- Un **algoritmo de planificación**, que define un criterio (política de planificación) que determina el orden de acceso de las tareas a los distintos procesadores.
- Un **método de análisis** (test de planificabilidad) que permite predecir el comportamiento temporal del sistema, y determina si la planificabilidad es factible bajo las condiciones o restricciones especificadas

  - Se pueden comprobar si los requisitos temporales están garantizados en **todos los casos posibles**.
  - En general se estudia el **peor comportamiento posible**, es decir, con el WCET (*Worst Case Execution Time*).

Cálculo del WCET.

- Todos los métodos de planificación parte de que se conoce el WCET (C), el tiempo de peor ejecución de cada tarea.
- ¿ Cómo podemos calcular el valor de C para cada tarea ?
- Hay dos formas de obtenerlo:
  - Medida directa del tiempo de ejecución (en el peor caso) en la plataforma de ejecución.
    - Se realizan múltiples experimentos, y se hace una estadística.
  - Análisis del código ejecutable
    - Se descompone el código en un grafo de bloques secuenciales.
    - Se calcula el tiempo de ejecución de cada bloque.
    - Se busca el camino más largo.

Restricciones temporales

Para determinar la planificación del sistema necesitamos conocer las **restricciones temporales** de cada tarea del sistema.
Aquí vemos dos ejemplos de restricciones temporales:

- $C = 2\,\text{ms}$, $T = D = 10\,\text{ms}$. Es una tarea periódica que se activa cada 10 ms, y se ejecuta en un tiempo máximo de 2 ms en el peor de los casos.
- $C = 10\,\text{ms}$, $T = 100\,\text{ms}$; $D = 90\,\text{ms}$. Es una tarea periódica que se activa cada 100 ms, se ejecuta en cada activación un máximo de 10 ms, y desde que se inicia la activación hasta que concluye no puede pasar más de 90 ms.

5.2 Esquemas de planificación

Tipos de Esquemas de Planificación

Los esquemas de planificación para un sistema monoprocesador son:

- **Planificación estática *off-line* sin prioridades**
  - Planificación cíclica (ejecutivo cíclico)
- **Planificación basada en prioridades**
  - Prioridades estáticas

  - Prioridad al más frecuente (RMS, *Rate Monotonic Scheduling*)
  - Prioridad al más urgente (DMS, *Deadline Monotonic Scheduling*)
- Prioridades dinámicas
  - Proximidad del plazo de respuesta (EDF, *Earliest Deadline First*)
  - Prioridad al de menor holgura (LLF, *Least Laxity First*)

5.2.1 Planificación Cíclica.

Planificación Cíclica

- La planificación se basa en construir un plan de ejecución (**plan principal**) de forma explicita y fuera de línea (*off-line*) en el que se especifica el entrelazado de las tareas periódicas de tal forma que su ejecución cíclica garantiza el cumplimiento de los plazos de las tareas.
  - La estructura de control o programa cíclico se denomina **ejecutivo cíclico**.
- El entrelazado es fijo y se define en el **plan principal** que se construye antes de poner en marcha el sistema (*off-line*).
- La duración del ciclo principal es igual al hiperperiodo
  - $T_M = \text{mcm}(T_1, T_2, \ldots, T_n)$
  - se supone tiempo entero (ticks de reloj)
  - el comportamiento temporal del sistema se repite cada ciclo principal

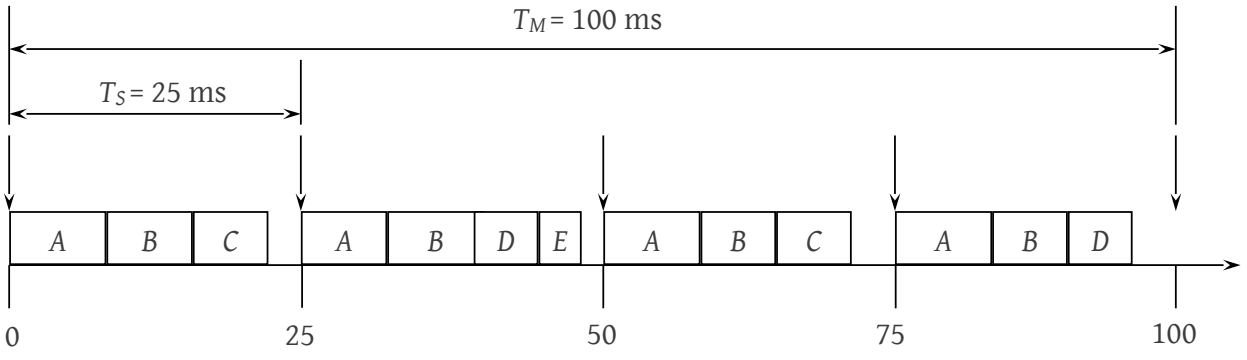
Ejemplo de planificación cíclica.

Tarea	T	C
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

- El ciclo principal dura 100 ms

$$T_M = \text{mcm}(25, 25, 50, 50, 100) = 100$$

- Se compone de 4 ciclos secundarios de 25 ms cada uno.



Implementación de la Planificación Cíclica

Esta planificación se puede implementar así:

```
void ejecutivo_ciclico()
{
   int nciclos = 4 ,
       frame   = 0 ;

   while( true )
   {
      switch( frame )
      {
         case 0 : A; B; C;     break ;
         case 1 : A; B; D; E;  break ;
         case 2 : A; B; C;     break ;
         case 3 : A; B; D;     break ;
      }
      frame = ( frame+1 ) % nciclos;
      esperar_tick_reloj( 25 ) ; /* espera inicio siguiente periodo de 25 miliseg. */
   }
} // final del ejecutivo ciclico
```

Propiedades de la Planificación Cíclica

- No hay concurrencia en la ejecución.
  - Cada ciclo secundario es una secuencia de llamadas a **procedimientos**
  - No se necesita un núcleo de ejecución multitarea
- Los procedimientos pueden **compartir datos**.
  - No se necesitan mecanismos de exclusión mutua como los semáforos o monitores
- No hace falta analizar el comportamiento temporal.
  - El sistema es correcto por construcción.

Problemas de la Planificación Cíclica

- Dificultad para incorporar tareas con periodos largos.
- Las tareas esporádicas son difíciles de tratar.
  - Se puede utilizar un servidor de consulta.
- El plan ciclo del proyecto es difícil de construir.
  - Si los periodos son de diferentes órdenes de magnitud el número de ciclos secundarios se hace muy grande.

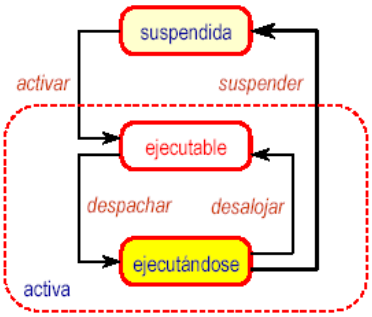
  - Puede ser necesario partir una tarea en varios procedimientos.
  - Es el caso más general de sistemas en tiempo real críticos.
- Es poco flexible y difícil de mantener.
  - Cada vez que se cambia una tarea hay que rehacer toda la planificación.

5.2.2 Planificación con prioridades

Planificación con prioridades

- La prioridad es un mecanismo elemental para planificar la ejecución de un conjunto de tareas.
  - Es un atributo de las tareas normalmente ligado a su importancia relativa en el conjunto de tareas.
  - Por convención se asigna números enteros mayores a procesos más urgentes.
  - La prioridad de una tarea la determina sus necesidades temporales; no es importante el rendimiento o comportamiento del sistema.

- Una tarea puede estar en varios estados:
- Las tareas ejecutables se despachan para su ejecución en orden de prioridad.



Planificación RMS

- RMS (***Rate Monotonic Scheduling***): Es un método de planificación estático on-line con asignación de prioridades a las tareas más frecuentes.
- A cada tarea i se le asigna una (única) prioridad P_i basada en su período (T_i): cuanto menor sea el periodo (mayor frecuencia) → mayor prioridad.

$$\forall i,j \ : \quad T_i < T_j \implies P_i > P_j$$

- Esta asignación de prioridades es optima en el caso de que todas las tareas sean periódicas, y el plazo de respuesta máxima D coincida con el periodo.

Ejemplo de planificación RMS (1)

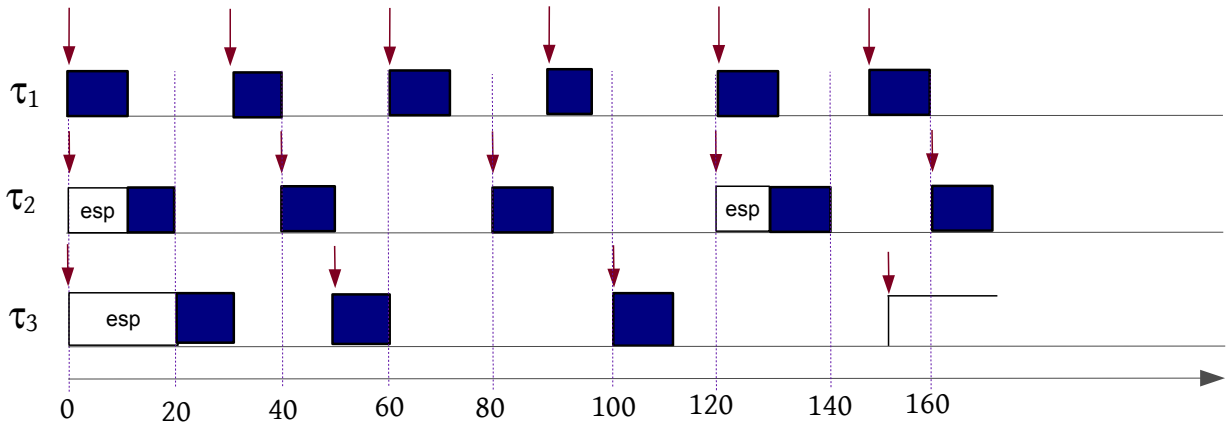
- Dada un conjunto de tres tareas con las siguientes restricciones temporales:

Tareas	C	D	T
1	10	30	30
2	5	40	40
3	9	50	50

- La aplicación del método de planificación RMS indica que la tarea 1 es la que tiene mayor prioridad, luego la 2, y luego la 3.

Ejemplo de planificación RMS (2)

- Para analizar la planificabilidad del sistema con dichas restricciones, hay que estudiar en el cronograma que:
  - Para cada tarea: $\rightarrow$ $R_i < T_i$
- Solo hay que probar la ejecución correcta de todas las tareas en el hiperperiodo, es decir, $H = \text{mcm}(Ti)$.
  - Si se cumple en el hiperperiodo, se repetirá indefinidamente.

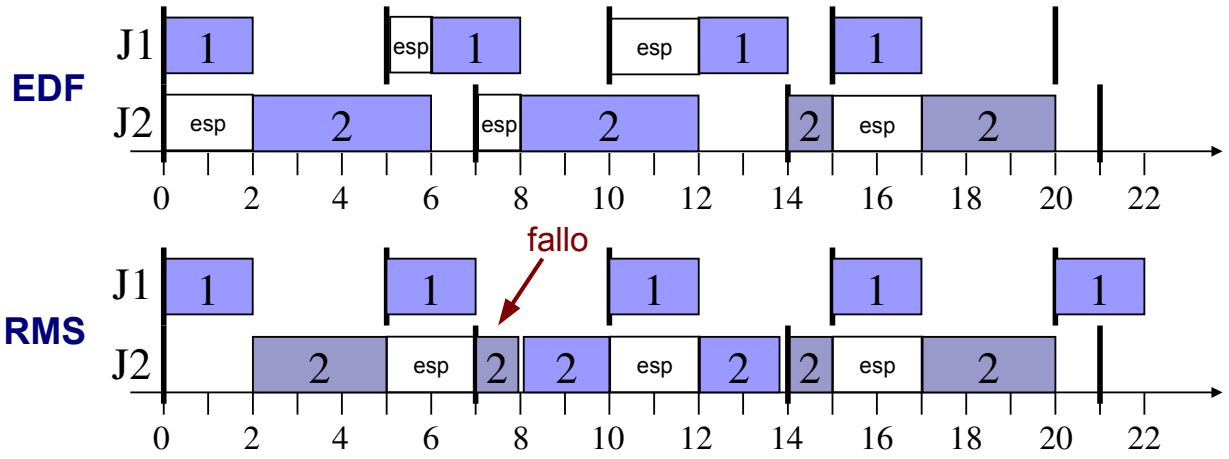


Planificación EDF

- EDF (*Earliest Deadline First*, primero al más urgente): La asignación de prioridad se establece una prioridad más alta a la que se encuentre más próxima a su plazo de respuesta máxima (deadline). En caso de igualdad se hace una elección no determinista de la siguiente tarea a ejecutar.
- Es un algoritmo de planificación dinámica, dado que la prioridad de cada tarea cambia durante la evolución del sistema.
- Es más óptimo porque no es necesario que las tareas sean periódicas.
- En menos pesimista que RMS.

Ejemplo de planificación EDF

Tareas	C	T	D
J1	2	5	5
J2	4	7	7



Part III

Seminarios y guiones de prácticas

Chapter 6

Seminario 1. Programación multihebra y sincronización con semáforos.

Introducción

Este seminario tiene cuatro partes, inicialmente se repasa el concepto de hebra, a continuación se da una breve introducción a la interfaz (posix) de las librerías de hebras disponibles en linux. A continuación, se estudia el mecanismo de los semáforos como herramienta para solucionar problemas de sincronización y, por último, se hace una introducción a una librería para utilizar semáforos con hebras posix.

- El objetivo es conocer algunas llamadas básicas de dicho interfaz para el desarrollo de ejemplos sencillos de sincronización con hebras usando semáforos (práctica 1)
- Las partes relacionadas con hebras posix están basadas en el texto disponible en esta web: https://computing.llnl.gov

6.1 Concepto e Implementaciones de Hebras

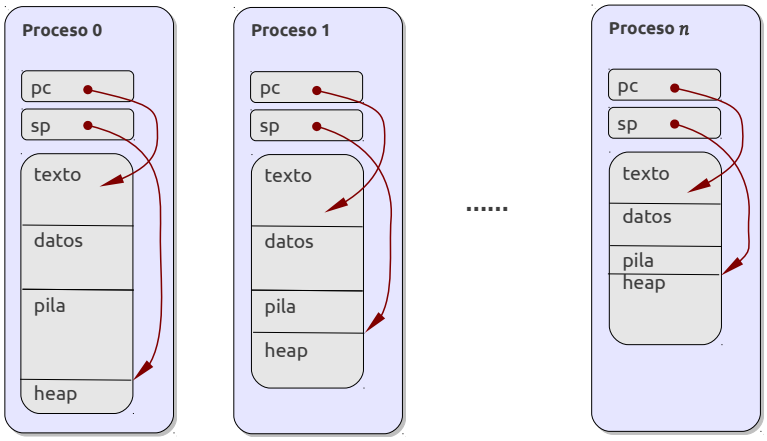
Procesos: estructura

En un sistema operativo pueden existir muchos procesos ejecutándose concurrentemente, cada proceso corresponde a un programa en ejecución y tiene su propio flujo de control.

- Cada proceso ocupa una zona de memoria con (al menos) estas partes:
  - **texto**: zona con la secuencia de instrucciones que se están ejecutando.
  - **datos**: espacio (de tamaño fijo) ocupado por variables globales.
  - **pila**: espacio (de tamaño cambiante) ocupado por variables locales.
  - **mem. dinámica** (***heap***): espacio ocupado por variables dinámicas.
- Cada proceso tiene asociados (entre otros) estos datos:
  - **contador de programa** (**pc**): dir. en memoria (zona de texto) de la siguiente instrucción a ejecutar.
  - **puntero de pila** (**sp**): dir. en memoria (zona de pila) de la última posición ocupada por la pila.

Diagrama de la estructura de los procesos

Podemos visualizarla (simplificadamente) como sigue:

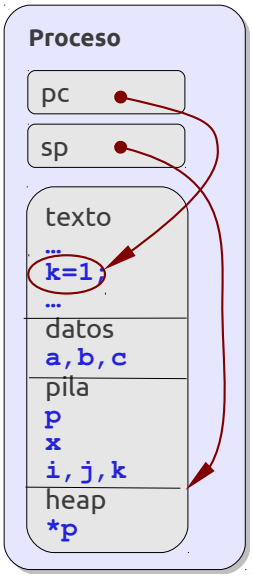


Ejemplo de un proceso

En el siguiente programa escrito en C/C++, el estado del proceso (durante la ejecución de k=1;) es el que se ve a la derecha:

```
int a,b,c ; // variables globales

void subprograma1()
{
   int i,j,k ; // vars. locales (1)
   k = 1 ;
}
void subprograma2()
{
   float x ; // vars. locales (2)
   subprograma1() ;
}
int main()
{
   char * p = new char ; // "p" local
   *p = 'a'; // "*p" en el heap
   subprograma2() ;
}
```



Procesos y hebras

La gestión de varios procesos no independientes (cooperantes) es muy útil pero consume una cantidad apreciable de recursos del SO:

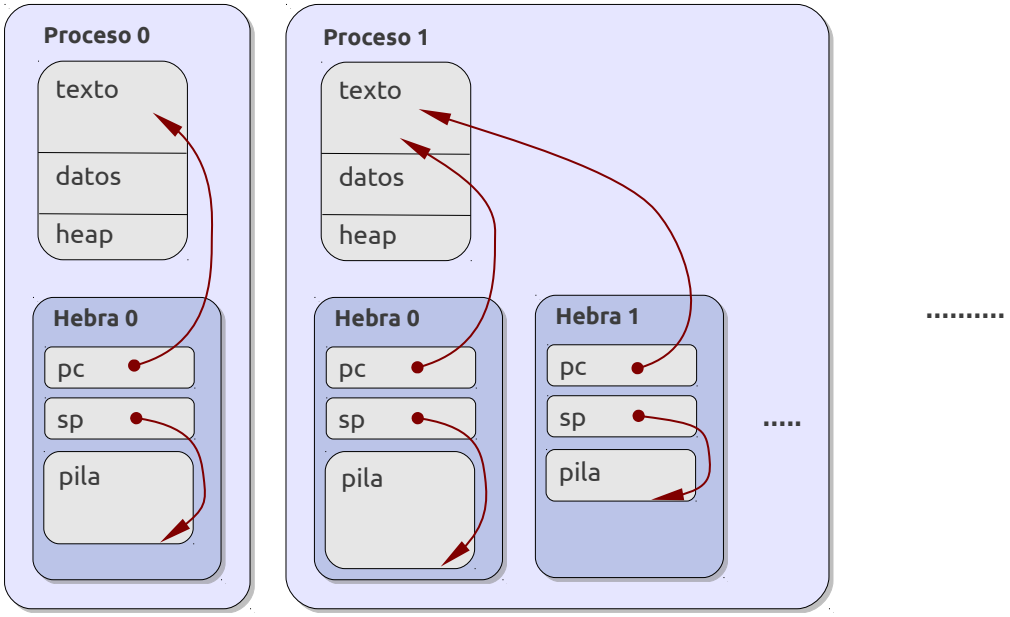
- Tiempo de procesamiento para repartir la CPU entre ellos
- Memoria con datos del SO relativos a cada proceso
- Tiempo y memoria para comunicaciones entre esos procesos

para mayor eficiencia en esta situación se diseñó el concepto de **hebra**:

- Un proceso puede contener una o varias hebras.
- Una hebra es un flujo de control en el texto (común) del proceso al que pertencen.
- Cada hebra tiene su propia pila (vars. locales), vacía al inicio.
- Las hebras de un proceso comparten la zona de datos (vars. globales), y el *heap*.

Diagrama de la estructura de procesos y hebras

Podriamos visualizarlos (simplificadamente) como sigue:



Inicio y finalización de hebras

Al inicio de un programa, existe una única hebra (que ejecuta la función **main** en C/C++). Durante la ejecución del programa:

- Una hebra A en ejecución puede crear otra hebra B en el mismo proceso de A.
- Para ello, A designa un subprograma f (una función C/C++) del texto del proceso, y después continúa su ejecución. La hebra B:

  - ejecuta el subprograma f concurrentemente con el resto de hebras.
  - termina normalmente cuando finaliza de ejecutar dicho subprograma
- Una hebra puede finalizar en cualquier momento (antes de terminar el subprograma con que se inició).
- Una hebra puede finalizar otra hebra en ejecución B, sin esperar que termine
- Una hebra puede esperar a que cualquier otra hebra en ejecución finalice.

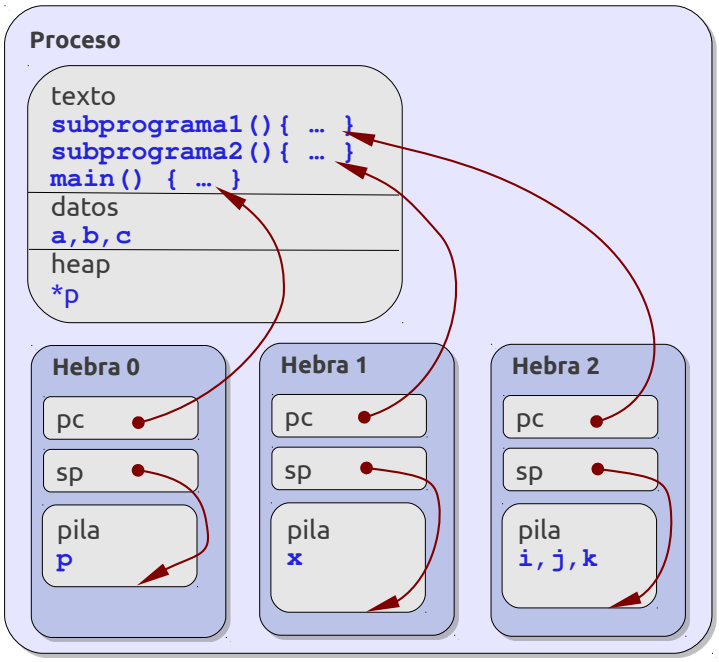
Ejemplo de estado de un proceso con tres hebras

En **main** se crean dos hebras, después se llega al estado que vemos:

```
int a,b,c ;

void subprograma1()
{
   int i,j,k ;
   // ...
}
void subprograma2()
{
   float x ;
   // ...
}

int main()
{
   char * p = new char ;
   // cr.he.subp.2
   // cr.he.subp.1
   // ...
}
```



6.2 Hebras POSIX

6.2.1 Introducción

Introducción

En esta sección veremos algunas llamadas básicas de la parte de hebras de POSIX (IEEE std 1003.1), en la versión de 2004:

- El estándar POSIX define los parámetros y la semántica de un amplio conjunto de funciones para diversos servicios del SO a los programas de usuario.
  - http://pubs.opengroup.org/onlinepubs/009695399/

- Una parte de las llamadas de POSIX están dedicadas a gestión (creación, finalización, sincronización, etc...) de hebras, son las que aparecen aquí:
  - http://pubs.opengroup.org/onlinepubs/009695399/idx/threads.html

La librería NPTL

Los ejemplos se han probado usando la *Native POSIX Thread Library* (NTPL) en Linux (ubuntu), que implementa la parte de hebras de la versión de 2004 de POSIX

- Está disponible para Linux desde el kernel 2.6 (2004).
- En esta implementación una hebra POSIX se implementa usando una *hebra del kernel* de Linux (se dice que las hebras POSIX son 1-1).
- Como consecuencia de lo anterior, hebras distintas de un mismo proceso pueden ejecutarse en procesadores distintos,
- Por tanto, este tipo de hebras constituyen una herramienta ideal para el desarrollo de aplicaciones que pueden aprovechar el potencial de rendimiento que ofrecen los sistemas multiprocesador (o multinúcleo) con memoria compartida.

6.2.2 Creación y finalización de hebras

Creación de hebras con *pthread_create*

Esta función sirve para crear una nueva hebra, su declaración es como sigue:

```
int pthread_create( pthread_t *thread, const pthread_attr_t *attr,
   void *(*start_routine)(void*), void *arg);
```

Parámetros (más información aquí):

nombre	tipo	descripción
thread	pthread_t *	para referencias posteriores en las operaciones sobre la hebra
attr	pthread_attr_t *	atributos de la hebra (puede ser NULL para valores por def.)
start_routine	void *nombre(void *)	función a ejecutar por la hebra
arg	void *	puntero que se pasa como parámetro para start_routine (puede ser NULL).

Ejemplo de creación de hebras.

En el siguiente ejemplo se crean dos hebras:

```
#include <iostream>
#include <pthread.h>
using namespace std ;
```

```
void* proc1( void* arg )
{  for( unsigned long i = 0 ; i < 5000 ; i++ )
      cout << "hebra 1, i == " << i << endl ;
   return NULL ;
}
void* proc2( void* arg )
{  for( unsigned long i = 0 ; i < 5000 ; i++ )
      cout << "hebra 2, i == " << i << endl ;
   return NULL ;
}
int main()
{
   pthread_t hebra1, hebra2 ;
   pthread_create(&hebra1,NULL,proc1,NULL);
   pthread_create(&hebra2,NULL,proc2,NULL);
   // ... finalizacion ....
}
```

Finalizacion de hebras

Una hebra cualquiera A finaliza cuando:

- A acaba de ejecutar la función f que se designó al crearla, de dos formas posibles:
  - A ejecuta un **return** en f
  - A llega al final de f
- A llama explícitamente a **pthread_exit** durante su ejecución.
- otra hebra B llama a **pthread_cancel**(A) (la hebra B *mata* a la hebra A)

Todas las hebras de un programa finalizan cuando:

- Cualquiera de ellas llama a **exit**
- Termina de ejecutarse la hebra principal sin haber llamado a **pthread_exit**. (por tanto, si la hebra principal debe acabar mientras pueden continuar las demás, es necesario que la principal acabe llamando a **pthread_exit**).

La función *pthread_exit*

La función **pthread_exit** causa la finalización de la hebra que la llama:

```
void pthread_exit( void* value_ptr );
```

Parámetros:

nombre	tipo	descripción
value_ptr	void *	puntero que recibirá la hebra que espere (vía *join*) la finalización de esta hebra (si hay alguna) (puede ser NULL)

más información:

http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_exit.html

Ejemplo de *pthread_exit*

El ejemplo anterior se puede completar así:

```
#include <iostream>
#include <pthread.h>
using namespace std ;

void* proc1( void* arg )
{  for( unsigned long i = 0 ; i < 5000 ; i++ )
      cout << "hebra 1, i == " << i << endl ;
   return NULL ;
}
void* proc2( void* arg )
{  for( unsigned long i = 0 ; i < 5000 ; i++ )
      cout << "hebra 2, i == " << i << endl ;
   return NULL ;
}
int main()
{
   pthread_t hebra1, hebra2 ;
   pthread_create(&hebra1,NULL,proc1,NULL);
   pthread_create(&hebra2,NULL,proc2,NULL);
   pthread_exit(NULL); // permite continuar a hebra1 y hebra2
}
```

6.2.3 Sincronización mediante unión

La operación de unión.

POSIX provee diversos mecanismos para sincronizar hebras, veremos dos de ellos:

- Usando la operación de *unión* (*join*).
- Usando semáforos.

La operación de unión permite que (mediante una llamada a **pthread_join**) una hebra A espere a que otra hebra B termine:

- A es la hebra que invoca la unión, y B la hebra *objetivo*.
- Al finalizar la llamada, la hebra objetivo ha terminado con seguridad.
- Si B ya ha terminado, no se hace nada.
- Si la espera es necesaria, se produce sin que la hebra que llama (A) consuma CPU durante dicha espera (A queda suspendida).

La función *pthread_join*

La función **pthread_join** está declarada como sigue:

```
int pthread_join( pthread_t thread, void **value_ptr );
```

Parámetros y resultado:

nombre	tipo	descripción
thread	**thread_t**	identificador de la hebra objetivo
value_ptr	**void ****	puntero a la variable que recibirá el dato (de tipo **void ***) enviado por la hebra objetivo al finalizar (vía **return** o **pthread_exit**)
resultado	**int**	0 si no hay error, en caso contrario se devuelve un código de error.

más información aquí: http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_join.html

Ejemplo de *pthread_join*

Puede ser útil, por ejemplo, para que la hebra principal realice algún procesamiento posterior a la finalización de las hebras:

```
#include <pthread.h>

void* proc1( void* arg ) { /* .... */ }
void* proc2( void* arg ) { /* .... */ }

int main()
{
   pthread_t hebra1, hebra2 ;

   pthread_create(&hebra1,NULL,proc1,NULL);
   pthread_create(&hebra2,NULL,proc2,NULL);

   pthread_join(hebra1,NULL);
   pthread_join(hebra2,NULL);

   // calculo posterior a la finalizacion de las hebras.....
}
```

6.2.4 Parámetros e identificación de hebras

Hebras idénticas

En muchos casos, un problema se puede resolver con un proceso en el que varias hebras distintas ejecutan el mismo algoritmo con distintos datos de entrada. En estos casos

- Es necesario que cada hebra reciba parámetros distintos
- Esto se puede hacer a traves del parámetro de tipo **void *** que recibe el subprograma que ejecuta una hebra.

- Dicho parámetro suele ser un índice o un puntero que permita a la hebra recuperar el conjunto de parámetros de entrada de una estructura de datos en memoria compartida, inicializada por la hebra principal (si es un índice entero, es necesario convertirlo hacia/desde el tipo **void** *)
- La estructura puede usarse para guardar también resultados de la hebra, o en general datos de la misma que deban ser leídos por otras.

Ejemplo básico de paso de parámetros a hebras

10 hebras ejecutan concurrentemente **fun_hebra(0)**, **fun_hebra(1)**, ...,

```
#include <iostream>
#include <pthread.h>
using namespace std ;
const unsigned num_hebras = 10 ;

// función que ejecuta cada hebra (cálculo arbitrario)
void* fun_hebra( void* arg_ptr )
{
  unsigned long arg_ent = (unsigned long)arg_ptr ; // convertir puntero en entero
  unsigned long res_ent ;                          // resultado (entero)
  void *        res_ptr ;                          // resultado (puntero)

  // aquí se incluye cualquier cálculo que le asigne
  // un valor a res_ent usando el valor de arg_ent, por ejemplo este:
  res_ent = arg_ent*arg_ent + 5*arg_ent + 2 ;

  // valor ya calculado: devolver resultado como un puntero
  res_ptr = (void *) res_ent ;   // convertir entero en puntero
  return res_ptr ;               // devolver resultado (puntero)
}
...
```

nota: solo funciona si **sizeof(unsigned long)**==**sizeof(void *)**, es lo usual.

Ejemplo de paso de parámetros (2)

```
....
int main()
{
   pthread_t id_hebra[num_hebras] ; // vector de identificadores de hebra

   // lanzar hebras
   for( unsigned long i = 0 ; i < num_hebras ; i++ )
   {   void * arg_ptr = (void *) i ; // convertir entero a puntero
       pthread_create( &(id_hebra[i]), NULL, fun_hebra, arg_ptr );
   }

   unsigned long res_ent ;   // guardará el resultado de cada hebra
   void ** ptr_res_ent = (void **) &res_ent; // puntero a var. con result.
```

```
   // esperar hebras e imprimir resultados
   for( unsigned i = 0 ; i < num_hebras ; i++ )
   {  pthread_join( id_hebra[i], ptr_res_ent );   // asigna a res_ent
      cout << "func(" <<  i << ") == " << res_ent << endl  ;
   }
}
```

6.2.5 Ejemplo de hebras: cálculo numérico de integrales

Cálculo de numérico de integrales

La programación concurrente puede ser usada para resolver más rápidamente multitud de problemas, entre ellos los que conllevan muchas operaciones con números flotantes

- Un ejemplo típico es el cálculo del valor I de la integral de una función f de variable real (entre 0 y 1, por ejemplo) y valores reales positivos:

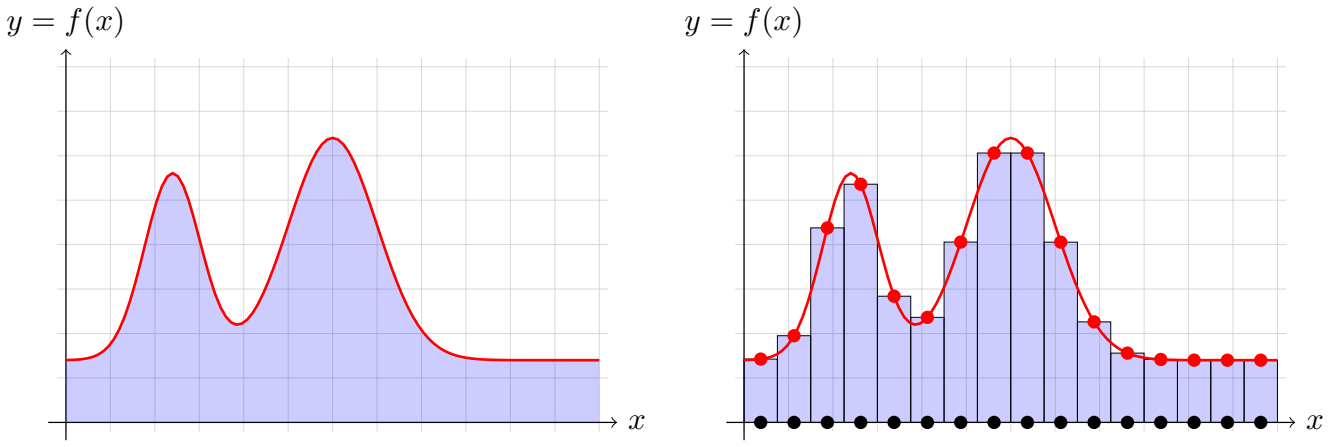
$$I = \int_0^1 f(x)\,dx$$

- El cálculo se puede hacer evaluando la función f en un conjunto de m puntos uniformemente espaciados en el intervalo $[0,1]$, y aproximando I como la media de todos esos valores:

$$I \approx \frac{1}{m}\sum_{i=0}^{m-1} f(x_i) \qquad \text{donde:} \quad x_i = \frac{i+1/2}{m}$$

Interpretación geométrica

Aproximamos el área azul (es I) (izquierda), usando la suma de las áreas de las m barras (derecha):



- Cada punto de muestra es el valor x_i (puntos negros)
- Cada barra tiene el mismo ancho $1/m$, y su altura es $f(x_i)$.

Cálculo secuencial del número π

Para verificar la corrección del método, se puede usar una integral I con valor conocido. A modo de ejemplo, usaremos una función f cuya integral entre 0 y 1 es el número π:

$$I = \pi = \int_0^1 \frac{4}{1+x^2}\,dx \qquad \text{aquí } f(x) = \frac{4}{1+x^2}$$

una implementación secuencial sencilla sería mediante esta función:

```
unsigned long m = ..... ;   // número de muestras

double f( double x )        // implementa función f
{ return 4.0/(1+x*x) ;      //    f(x) = 4/(1+x^2)
}
double calcular_integral_secuencial( )
{  double suma = 0.0 ;                          // inicializar suma
   for( unsigned long i = 0 ; i < m ; i++ )     // para cada i entre 0 y m-1
      suma += f( (i+0.5)/m );                   //      añadir f(x_i) a la suma actual
   return suma/m ;                              // devolver valor promedio de f
}
```

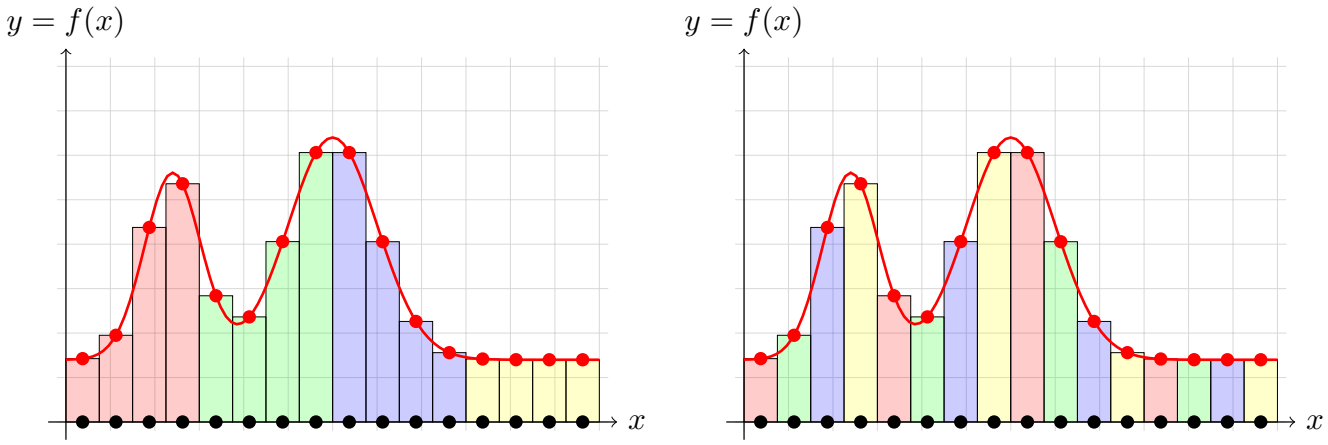
Versión concurrente de la integración

El cálculo citado anteriormente se puede hacer mediante un total de n hebras idénticas (asumimos que m es múltiplo de n)

- Cada una de las hebras evalua f en m/n puntos del dominio
- La cantidad de trabajo es similar para todas, y los cálculos son independientes.
- Cada hebra calcula la suma parcial de los valores de f
- La hebra principal recoge las sumas parciales y calcula la suma total.
- En un entorno con k procesadores o núcleos, el cálculo puede hacerse hasta k veces más rápido. Esta mejora ocurre solo para valores de m varios órdenes de magnitud más grandes que n.

Distribución de cálculos

Para distribuir los cálculos entre hebras, hay dos opciones simples, hacerlo de forma **contigua** (izquierda) o de forma **entrelazada** (derecha)



Cada valor $f(x_i)$ es calculado por:

- **Contigua:** la hebra número i/n.
- **Entrelazada:** la hebra número $i \mod n$.

Esquema de la implementación concurrente

```
const unsigned long m = .... ; // número de muestras
const unsigned long n = .... ; // número de hebras
double resultado_parcial[n] ;  // vector de resultados parciales

double f( double x )           // implementa función f:
{ return 4.0/(1+x*x) ;         //    f(x) = 4/(1+x^2)
}
void * funcion_hebra( void * ih_void ) // función que ejecuta cada hebra
{
   unsigned long ih = (unsigned long) ih_void ; // número o índice de esta hebra
   double sumap = 0.0 ;
   // calcular suma parcial en "sumap"
   .....
   resultado_parcial[ih] = sumap ; // guardar suma parcial en vector.
}
double calcular_integral_concurrente( )
{
   // crear y lanzar n hebras, cada una ejecuta "funcion_hebra"
   .....
   // esperar (join) a que termine cada hebra, sumar su resultado
   .....
   // devolver resultado completo
   .....
}
```

Medición de tiempos

Para apreciar la reducción de tiempos que se consigue con la programación concurrente, se pueden medir los tiempos que tardan la versión secuencial y la concurrente. Para hacer esto con la máxima precisión, se puede usar la función `clock_gettime` en linux:

- Forma parte de POSIX, en concreto de la extensiones de tiempo real.
- Sirve para medir tiempo real transcurrido entre instantes de la ejecución de un programa con muy alta precisión (nanosegundos).
- Para facilitar su uso, en lugar de usarla directamente se pueden usar indirectamente a través de otras funciones (que se proporcionan) que permiten calcular ese tiempo como un valor real en segundos, haciendo abstracción de los detalles no relevantes.

Uso de las funciones de medición de tiempos

Para medir el tiempo que tarda un trozo de programa, se puede usar este esquema en C++

```
#include "fun_tiempos.h"
// ....
void funcion_cualquiera( /*.....*/ )
{  .....
   struct timespec inicio = ahora() ;  // inicio = inicio del tiempo a medir
   ....                                // actividad cuya duracion se quiere medir
   struct timespec fin = ahora() ;     // fin = fin del tiempo a medir
   ....
   cout << "tiempo transcurrido == "  // escribe resultados:
        << duracion( &inicio, &fin )  // tiempo en segundos entre "inicio" y "fin"
        << " seg." << endl ;
}
```

los archivos `fun_tiempo.h` (cabeceras) y `fun_tiempo.c` (implementación) se encuentran disponibles para los alumnos.

Compilando programas con hebras POSIX

Para compilar un archivo `ejemplo.cpp`, que use las funciones definidas en `fun_tiempos.c` (y use hebras posix), y obtener el archivo ejecutable `ejemplo`, podemos dar estos pasos:

```
gcc -g -c fun_tiempos.c    # compila "fun_tiempos.c" y genera "fun_tiempos.o"
g++ -g -c ejemplo.cpp      # compila "ejemplo.cpp" y genera "ejemplo.o"
g++ -o ejemplo ejemplo.o fun_tiempos.o -lrt -lpthread # enlaza, genera "ejemplo"
```

- el *switch* `-lrt` sirve para enlazar las librerías correspondientes a la extensión de tiempo real de POSIX (incluye `clock_gettime`)
- el *switch* `-lpthreads` sirve para incluir las funciones de hebras POSIX
- también es posible integrarlo todo en un `makefile` y usar `make`

Actividad: medición de tiempos de cálculo concurrente.

Como actividad para los alumnos en este seminario se propone realizar y ejecutar una implementación sencilla del cálculo concurrente del número π, tal y como hemos visto aquí:

- El programa aceptará como parámetros en la línea de comandos el número de hebras a lanzar y el número de muestras
- En la salida se presenta el valor exacto de π y el calculado (sirve para verificar si el programa es correcto, ya que deben diferir por muy poco para un número de muestras del orden de cientos o miles)
- Asimismo , el programa imprimirá la duración del cálculo concurrente y el secuencial.

Se debe razonar acerca de como el número de procesadores disponibles y el número de hebras afecta al tiempo del cálculo concurrente en relación al secuencial (en Linux, para conocer el número de CPUs disponibles y sus características, se puede ver el archivo `/proc/cpuinfo`)

6.3 Introducción a los Semáforos

Semáforos

Los **semáforos** constituyen un mecanismo de nivel medio que permite solucionar los problemas derivados de la ejecución concurrente de procesos no independientes. Sus características principales son:

- permite bloquear los procesos sin mantener ocupada la CPU
- resuelven fácilmente el problema de exclusión mutua con esquemas de uso sencillos
- se pueden usar para resolver problemas de sincronización (aunque en ocasiones los esquemas de uso son complejos)
- el mecanismo se implementa mediante instancias de una estructura de datos a las que se accede únicamente mediante subprogramas específicos.

Estructura de un semáforo

Un semáforo es un instancia de una estructura de datos (un registro) que contiene los siguientes elementos:

- Un conjunto de procesos bloqueados (se dice que están esperando en el semáforo).
- Un valor natural (entero no negativo), al que llamaremos *valor del semáforo*

Estas estructuras de datos residen en memoria compartida. Al principio de un programa que use semáforos, debe poder inicializarse cada uno de ellos:

- el conjunto de procesos asociados (bloqueados) estará vacío
- se deberá indicar un valor inicial del semáforo

Operaciones sobre los semáforos

Además de la inicialización, solo hay dos operaciones básicas que se pueden realizar sobre una variable de tipo semáforo (que llamamos s) :

- **sem_wait**(s)
  - Si el valor de s es mayor que cero, decrementar en una unidad dicho valor
  - Si el valor de s es cero, bloquear el proceso que la invoca en el conjunto de procesos bloqueados asociado a s
- **sem_signal**(s)
  - Si el conjunto de procesos bloqueados asociado a s no está vacío, desbloquear uno de dichos procesos.
  - Si el conjunto de procesos bloqueados asociado a s está vacío, incrementar en una unidad el valor de s.

En un semáforo cualquiera, estas operaciones se ejecutan de forma atómica, es decir, no puede haber dos procesos distintos ejecutando estas operaciones a la vez sobre un mismo semáforo (excluyendo el período de bloqueo que potencialmente conlleva la llamada a **sem_wait**).

Problema básico de sincronización

Un problema frecuente ocurre cuando:

- Un proceso **P2** no debe pasar de un punto de su código hasta que otro proceso **P1** no haya llegado a otro punto del suyo.
- El caso típico es: **P1** debe escribir una variable compartida y **después P2** debe leerla.

```
{ variables compartidas y valores iniciales }
var compartida : integer ; { variable compartida: P1 escribe y P2 lee }
```

```
process P1 ;
   var local1 : integer ;
begin
   .....
   local1 := ..... ;
   compartida := local1 ;
   .....
end
```

```
process P2 ;
   var local2 : integer ;
begin
   .....
   .....
   local2 := compartida ;
   .....
end
```

Este programa **no funciona correctamente**, por la hipótesis de progreso finito.

Solución con un semáforo

La solución es usar un semáforo, cuyo valor será:

- 1 si **P1** ha terminado de escribir, pero **P2** no ha comenzado a leer.
- 0 en cualquier otro caso (antes de terminar de escribir y después de comenzar a leer).

```
{ variables compartidas y valores iniciales }
var compartida  : integer ; { variable compartida: P1 escribe y P2 lee }
var puede_leer  : semaphore := 0 ; { 1 si la var. esta pte. de leer, 0 en otro caso }
```

```
process P1 ;
   var local1 : integer ;
begin
   .....
   local1 := ..... ;
   compartida := local1 ;
   sem_signal( puede_leer ) ;
   .....
end
```

```
process P2 ;
   var local2 : integer ;
begin
   .....
   .....
   .....
   sem_wait( puede_leer ) ;
   local2 := compartida ;
   .....
end
```

Uso de semáforos para exclusión mutua

Los semáforos se pueden usar para EM usando un semáforo inicializado a 1, y haciendo **sem_wait** antes de la sección crítica y **sem_signal** después de la sección crítica:

```
{ variables compartidas y valores iniciales }
var sc_libre : semaphore := 1 ; { 1 si SC esta libre, 0 si SC esta ocupada }

process ProcesosEM[ i : 0..n-1 ] ;
begin
   while true do begin
      sem_wait( sc_libre );    { esperar bloqueado hasta que 'sc_libre' sea 1 }
      { seccion critica:  ..... }
      sem_signal( sc_libre ); { desbl. proc. en espera o poner 'sc_libre' a 1 }
      { resto seccion:  ..... }
   end
end
```

En cualquier instante de tiempo, la suma del valor del semáforo más el número de procesos en la SC es la unidad. Por tanto, solo puede haber 0 o 1 procesos en SC, y se cumple la exclusión mutua.

Uso de semáforos para sincronización

El problema del Productor-Consumidor se puede resolver fácilmente con semáforos:

```
{ variables compartidas }
var
 x              : integer ;          { contiene cada valor producido      }
 puede_leer     : semaphore := 0 ;   { 1 si se puede leer "x", 0 si no    }
 puede_escribir : semaphore := 1 ;   { 1 si se puede escribir "x", 0 si no }
```

```
Process Productor ; { calcula "x" }
   var a : integer ;
begin
   while true begin
      a := ProducirValor() ;
      sem_wait( puede_escribir  );
      x := a ; { sentencia de escritura E }
      sem_signal( puede_leer ) ;
   end
end
```

```
Process Consumidor ; { lee "x" }
   var b : integer ;
begin
   while true do begin
      sem_wait( puede_leer ) ;
      b := x ; { sentencia de lectura L }
      sem_signal( puede_escribir ) ;
      UsarValor(b) ;
   end
end
```

6.4 Sincronización de hebras con semáforos POSIX

Introducción

Una parte de las llamadas del estándard POSIX son útiles para gestionar semáforos para sincronización de hebras o de procesos.

Veremos las llamadas básicas que permiten sincronizar hebras en un proceso. Son las siguientes:

- **sem_init**: inicializa un semáforo (dando el valor inicial)
- **sem_wait**: realiza la operación **wait** sobre un semáforo
- **sem_post**: realiza la operación **signal** sobre un semáforo
- **sem_destroy**: destruye un semáforo y libera la memoria ocupada.

6.4.1 Funciones básicas.

La función *sem_init*

La función **sem_init** está declarada como sigue:

```
int sem_init( sem_t* sem, int pshared, unsigned value );
```

Parámetros y resultado:

nombre	tipo	descripción
sem	**sem_t** *	puntero al identificador del semáforo
pshared	**int**	distinto de cero solo si el semáforo será compartido con otros procesos.
value	**unsigned**	valor inicial
resultado	**int**	0 si se ha inicializado el semáforo, en caso contrario es un código de error.

- Se debe usar **antes** de cualquier otra operación con el semáforo.

más información aquí: http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_init.html

La función *sem_wait*

La función **sem_wait** está declarada como sigue:

```
int sem_wait( sem_t* sem );
```

Parámetros y resultado:

nombre	tipo	descripción
sem	**sem_t** *	puntero al identificador del semáforo
resultado	**int**	0 solo si no ha habido error

más información aquí: http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_wait.html

La función *sem_post*

La función **sem_post** está declarada como sigue:

```
int sem_post( sem_t* sem );
```

Parámetros y resultado:

nombre	tipo	descripción
sem	**sem_t** *	puntero al identificador del semáforo
resultado	**int**	0 solo si no ha habido error

- La selección de la hebra a desbloquear (si hay alguna) depende de los parametros de *scheduling* asignados a la hebra (**normalmente será FIFO**).

Más información aquí: http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_post.html

La función *sem_destroy*

La función **sem_destroy** está declarada como sigue:

```
int sem_destroy( sem_t* sem ) ;
```

Parámetros y resultado:

nombre	tipo	descripción
sem	**sem_t** *	puntero al identificador del semáforo
resultado	**int**	0 solo si no ha habido error

- Solo se puede llamar para semáforos (inicializados con **sem_init**) en los cuales no haya hebras esperando.
- Después de destruir un semáforo, se puede volver a usar haciendo **sem_init**.

Más información aquí: http://pubs.opengroup.org/onlinepubs/009695399/functions/sem_destroy.html

6.4.2 Exclusión mutua

Ejemplo de exclusión mutua con semáforos POSIX

Los semáforos se pueden usar para exclusión mutua, por ejemplo,

En este ejemplo, se usa un semáforo (de nombre **mutex**) inicializado a 1, para escribir en la salida estándar una línea completa sin interrupciones.

```
#include <iostream>
#include <pthread.h>
#include <semaphore.h>
using namespace std ;

sem_t mutex ; // semaforo en memoria compartida

void* proc( void* p )
{
   sem_wait( &mutex );
      cout << "hebra numero: " << ((unsigned long) p) << ". " << endl ;
   sem_post( &mutex );
   return NULL ;
}

....
```

Ejemplo de exclusión mutua con semáforos POSIX (2)

El procedimiento principal debe inicializar el semáforo y crear las hebras, como se incluye aquí:

```
...

int main()
{
   const unsigned num_hebras = 50 ;
   pthread_t id_hebra[num_hebras] ;

   sem_init( &mutex, 0, 1 );

   for( unsigned i = 0 ; i < num_hebras ; i++ )
       pthread_create( &(id_hebra[i]), NULL, proc, (void *)i );

   for( unsigned i = 0 ; i < num_hebras ; i++ )
       pthread_join( id_hebra[i], NULL );
```

```
   sem_destroy( &mutex );

}
```

6.4.3 Sincronización

Ejemplo de sincronización con semáforos POSIX

En este otro ejemplo, hay una hebra que escribe una variable global y otra que la lee (cada una en un bucle). Se usan dos semáforos para evitar dos lecturas o dos escrituras seguidas, y además otro para exclusión mútua en las escrituras al terminal:

```
sem_t
   puede_escribir, // inicializado a 1
   puede_leer,     // inicializado a 0
   mutex ;         // inicializado a 1

unsigned long
   valor_compartido ; // valor para escribir o leer

const unsigned long
   num_iter = 10000 ; // numero de iteraciones

......
```

Ejemplo de sincronización con semáforos POSIX (2)

La hebra escritora espera que se pueda escribir, entonces escribe y señala que se puede leer:

```
...

void* escribir( void* p )
{
   unsigned long contador = 0 ;

   for( unsigned long i = 0 ; i < num_iter ; i++ )
   {
      contador = contador + 1 ; // genera un nuevo valor
      sem_wait( &puede_escribir ) ;
      valor_compartido = contador ; // escribe el valor
      sem_post( &puede_leer ) ;

      sem_wait( &mutex ) ;
      cout << "valor escrito == " << contador << endl << flush ;
      sem_post( &mutex ) ;
   }
   return NULL ;
}
```

```
...
```

Ejemplo de sincronización con semáforos POSIX (3)

La hebra lectora espera a que se pueda leer, entonces lee y señala que se puede escribir:

```
....

void* leer( void* p )
{
   unsigned long  valor_leido ;

   for( unsigned long i = 0 ; i < num_iter ; i++ )
   {
      sem_wait( &puede_leer ) ;
      valor_leido = valor_compartido ; // lee el valor generado
      sem_post( &puede_escribir ) ;

      sem_wait( &mutex ) ;
      cout << "valor leido   == " << valor_leido << endl << flush ;
      sem_post( &mutex ) ;
   }
   return NULL ;
}

...
```

Ejemplo de sincronización con semáforos POSIX (4)

El procedimiento **main** crea los semáforos y hebras y espera que terminen:

```
....

int main()
{
   pthread_t hebra_escritora, hebra_lectora ;

   sem_init( &mutex,           0, 1 ); // semaforo para EM: inicializado a 1
   sem_init( &puede_escribir, 0, 1 ); // inicialmente se puede escribir
   sem_init( &puede_leer,      0, 0 ); // inicialmente no se puede leer

   pthread_create( &hebra_escritora, NULL, escribir, NULL );
   pthread_create( &hebra_lectora,   NULL, leer,     NULL );

   pthread_join( hebra_escritora, NULL ) ;
   pthread_join( hebra_lectora,   NULL ) ;

   sem_destroy( &puede_escribir );
   sem_destroy( &puede_leer );
   sem_destroy( &mutex );
}
```

Chapter 7

Práctica 1. Sincronización de hebras con semáforos.

7.1 Objetivos

Objetivos.

En esta práctica se realizarán dos implementaciones de dos problemas sencillos de sincronización usando librerías abiertas para programación multihebra y semáforos. Los objetivos son:

- Conocer el *problema del productor-consumidor* y sus aplicaciones.
  - Diseñar una solución al problema basada en semáforos.
  - Implementar esa solución en un programa C/C++ multihebra, usando la funcionalidad de la librería POSIX para:
    - la creación y destrucción de hebras
    - la sincronización de hebras usando semáforos
- Conocer un problema sencillo de sincronización de hebras (el *problema de los fumadores*)
  - Diseñar una solución basada en semáforos, teniendo en cuenta los problemas que pueden aparecer.
  - Implementar la solución a dicho problema en C/C++ usando hebras y semáforos POSIX.

7.2 El problema del productor-consumidor

7.2.1 Descripción del problema.

Problema y aplicaciones

El problema del productor consumidor surge cuando se quiere diseñar un programa en el cual un proceso o hebra produce items de datos en memoria que otro proceso o hebra consume.

- Un ejemplo sería una aplicación de reproducción de vídeo:

  - El **productor** se encarga de leer de disco o la red y descodificar cada cuadro de vídeo.
  - El **consumidor** lee los cuadros descodificados y los envía a la memoria de vídeo para que se muestren en pantalla

  hay muchos ejemplos de situaciones parecidas.

- En general, el productor calcula o produce una secuencia de items de datos (uno a uno), y el consumidor lee o consume dichos items (tambien uno a uno).
- El tiempo que se tarda en producir un item de datos puede ser variable y en general distinto al que se tarda en consumirlo (también variable).

Solución de dos hebras con un vector de items

Para diseñar un programa que solucione este problema:

- Suele ser conveniente implementar el productor y el consumidor como dos hebras independientes, ya que esto permite tener ocupadas las CPUs disponibles el máximo de tiempo,
- se puede usar una variable compartida que contiene un ítem de datos,
- las esperas asociadas a la lectura y la escritura pueden empeorar la eficiencia. Esto puede mejorarse usando un vector que pueda contener muchos items de datos producidos y pendientes de leer.

Condición de sincronización

En esta situación, la implementación debe asegurar que :

- cada ítem producido es leído (ningún ítem se pierde)
- ningún ítem se lee más de una vez.

lo cual implica:

- el productor tendrá que esperar antes de poder escribir en el vector cuando haya creado un ítem pero el vector esté completamente ocupado por ítems pendientes de leer
- el consumidor debe esperar cuando vaya a leer un ítem del vector pero dicho vector no contenga ningún ítem pendiente de leer.
- en algunas aplicaciones el orden de lectura debe coincidir con el de escritura, en otras podría ser irrelevante.

7.2.2 Plantillas de código

Simplificaciones

En esta práctica se diseñará e implementará un ejemplo sencillo en C/C++

- cada ítem de datos será un valor entero de tipo int,

- el orden en el que se leen los items es irrelevante (en principio),
- el productor produce los valores enteros en secuencia, empezando en 1,
- el consumidor escribe cada valor leído en pantalla,
- se usará un vector intermedio de valores tipo int, de tamaño fijo pero arbitrario.

Funciones para producir y consumir:

Para producir un item de datos, la hebra productora invocará esta función:

```
int producir_dato()
{
    static int contador = 1 ;
    return contador ++ ;
}
```

mientras que la hebra consumidora llama a esta otra para consumir un dato:

```
void consumir_dato( int dato )
{
    cout << "dato recibido: " << dato << endl ;
}
```

Hebras productora y consumidora

Los subprogramas que ejecutan las hebras productora y consumidora son como se indica a continuación (no se incluye la sincronización ni los accesos al vector):

```
void * productor( void * )
{   for( unsigned i = 0 ; i < num_items ; i++ )
    {   int dato = producir_dato() ;
        // falta: insertar "dato" en el vector
    }
    return NULL ;
}
void * consumidor( void * )
{   for( unsigned i = 0 ; i < num_items ; i++ )
    {   int dato ;
        // falta: leer "dato" desde el vector intermedio
        consumir_dato( dato ) ;
    }
    return NULL ;
}
```

Es necesario definir la constante **num_items** con algún valor concreto (entre 50 y 100 es adecuado)

Gestión de la ocupación del vector intermedio

El vector intermedio (*buffer*) tiene una capacidad (número de celdas usables) fija prestablecidada en una constante del programa que llamamos, por ejemplo, `tam_vec`.

- La constente `tam_vec` deberá ser estrictamente menor que `num_items` (entre 10 y 20 sería adecuado).
- En cualquier instante de la ejecución, el número de celdas ocupadas en el vector (por items de datos producidos pero pendientes de leer) es un número entre 0 (el buffer estaría vacío) y `tam_vec` (el buffer estaría lleno).
- Además del vector, es necesario usar alguna o algunas variables adicionales que reflejen el estado de ocupación de dicho vector.
- Es necesario estudiar si el acceso a dicha variable o variables **requiere o no requiere sincronización alguna** entre el productor y el consumidor.

Soluciones para la gestión de la ocupación

Hay básicamente dos alternativas posibles para gestionar la ocupación, se detallan aquí:

- **LIFO** (pila acotada), se usa una única variable entera no negativa:
  - `primera_libre` = índice en el vector de la primera celda libre (inicialmente `0`). Esta variable se incrementa al escribir, y se decrementa al leer.
- **FIFO** (cola circular), se usan dos variables enteras no negativas:
  - `primera_ocupada` = índice en el vector de la primera celda ocupada (inicialmente `0`). Esta variable se incrementa al leer (módulo `tam_vector`).
  - `primera_libre` = índice en el vector de la primera celda libre (inicialmente `0`). Esta variable se incrementa al escribir (módulo `tam_vector`).

(asumismos que los índices del vector van desde `0` hasta `tam_vector-1`, ambos incluidos)

7.2.3 Actividades y documentación

Lista de actividades

Debes realizar las siguientes actividades en el orden indicado:

1. Diseña una solución que permita conocer qué entradas del vector están ocupadas y qué entradas están libres (usa alguna de las dos opciones dadas).
2. Diseña una solución, mediante semáforos, que permita realizar las esperas necesarias para cumplir los requisitos descritos.
3. Implementa la solución descrita en un programa C/C++ con hebras y semáforos POSIX, completando las plantillas incluidas en este guión. Ten en cuenta que el programa debe escribir la palabra **`fin`** cuando hayan terminado las dos hebras.
4. Comprueba que tu programa es correcto: verifica que cada número natural producido es consumido exactamente una vez.

Documentación a incluir dentro del portafolios

Se incorporará al portafolios un documento indicando la siguiente información:

1. Describe la variable o variables necesarias, y cómo se determina en qué posición se puede escribir y en qué posición se puede leer.
2. Describe los semáforos necesarios, la utilidad de los mismos, el valor inicial y en qué puntos del programa se debe usar **sem_wait** y **sem_signal** sobre ellos.
3. Incluye el código fuente completo de la solución adoptada.

7.3 El problema de los fumadores.

7.3.1 Descripción del problema.

Descripción del problema (1)

En este apartado se intenta resolver un problema algo más complejo usando hebras y semáforos POSIX.

Considerar un estanco en el que hay tres fumadores y un estanquero.

1.1. Cada fumador representa una hebra que realiza una actividad (fumar), invocando a una función **fumar()**, en un bucle infinito.

1.2. Cada fumador debe esperar antes de fumar a que se den ciertas condiciones (tener suministros para fumar), que dependen de la actividad del proceso que representa al estanquero.

1.3. El estanquero produce suministros para que los fumadores puedan fumar, también en un bucle infinito.

1.4. Para asegurar concurrencia real, es importante tener en cuenta que la solución diseñada **debe permitir que varios fumadores fumen simultáneamente**.

Descripción del problema (2)

A continuación se describen los requisitos para que los fumadores puedan fumar y el funcionamiento del proceso estanquero:

2.1. Antes de fumar es necesario liar un cigarro, para ello el fumador necesita tres ingredientes: tabaco, papel y cerillas.

2.2. Uno de los fumadores tiene papel y tabaco, otro tiene papel y cerillas, y otro tabaco y cerillas.

2.3. El estanquero selecciona **aleatoriamente un ingrediente** de los tres que se necesitan para hacer un cigarro, lo pone en el mostrador, desbloquea al fumador que necesita dicho ingrediente y después se bloquea, esperando la retirada del ingrediente.

2.4. El fumador desbloqueado toma el ingrediente del mostrador, desbloquea al estanquero para que pueda seguir sirviendo ingredientes y **después** fuma durante un tiempo aleatorio.

2.5. El estanquero, cuando se desbloquea, vuelve a poner un ingrediente aleatorio en el mostrador, y se repite el ciclo.

7.3.2 Plantillas de código

Simulación de la acción de fumar

Para simular la acción de fumar, **fumar()**, se puede usar la función **unsigned usleep(unsigned milisegundos)** que suspende a la hebra que la invoca tantos milisegundos como indica su único argumento. Para que el retardo sea aleatorio, se puede tomar como referencia el siguiente fragmento código:

```
#include <time.h>    // incluye "time(....)"
#include <unistd.h> // incluye "usleep(...)"
#include <stdlib.h> // incluye "rand(...)" y "srand"

// función que simula la acción de fumar
// como un retardo aleatorio de la hebra
void fumar()
{ // calcular un numero aleatorio de milisegundos (entre 1/10 y 2 segundos)
   const unsigned miliseg = 100U + (rand() % 1900U) ;
   usleep( 1000U*miliseg ); // retraso bloqueado durante miliseg milisegundos
}
int main()
{
  srand( time(NULL) ); // inicializa la semilla aleatoria
  // ....
}
```

7.3.3 Actividades y documentación.

Diseño de la solución

Diseña e implementa una solución al problema en C/C++ usando cuatro hebras y los semáforos necesarios. La solución debe cumplir los requisitos incluidos en la descripción, y además debe:

- Evitar interbloqueos entre las distintas hebras.
- Producir mensajes en la salida estándar que permitan hacer un seguimiento de la actividad de las hebras:
  - El estanquero debe indicar cuándo produce un suministro y qué suministro produce. Para establecer el ingrediente concreto (o, lo que es equivalente, directamente el fumador que podría usarlo), se debe usar también la función **rand()**.
  - Cada fumador debe indicar cuándo espera, qué producto espera, y cuándo comienza y finaliza de fumar.

Documentación a incluir dentro del portafolios

Se incorporará al portafolios un documento incluyendo los siguientes puntos:

1. Semáforos necesarios para sincronización y para cada uno de ellos:
   - Utilidad.

- Valor inicial.
- Hebras que hacen **sem_wait** y **sem_signal** sobre dicho semáforo.

2. Código fuente completo de la solución adoptada.

Chapter 8

Seminario 2. Hebras en Java.

Introducción

Este seminario es una breve introducción a la programación con hebras en el lenguaje Java.

- El objetivo es conocer las principales formas de crear y gestionar y hebras en Java, con objeto de desarrollar los ejemplos de la práctica 2.
- Se mostrarán las distintas formas de crear hebras en Java, los distintos estados de una hebra Java y cómo controlar la prioridad de las hebras.
- Se verá un mecanismo para asegurar la exclusión mutua en el acceso concurrente a bloques de código Java.

Enlaces para acceder a información complementaria

- Aprenda Java como si Estuviera en primero.
- Tutorial de Java.
- Manual de Java.
- Información sobre la clase `Thread`.

8.1 Hebras en Java

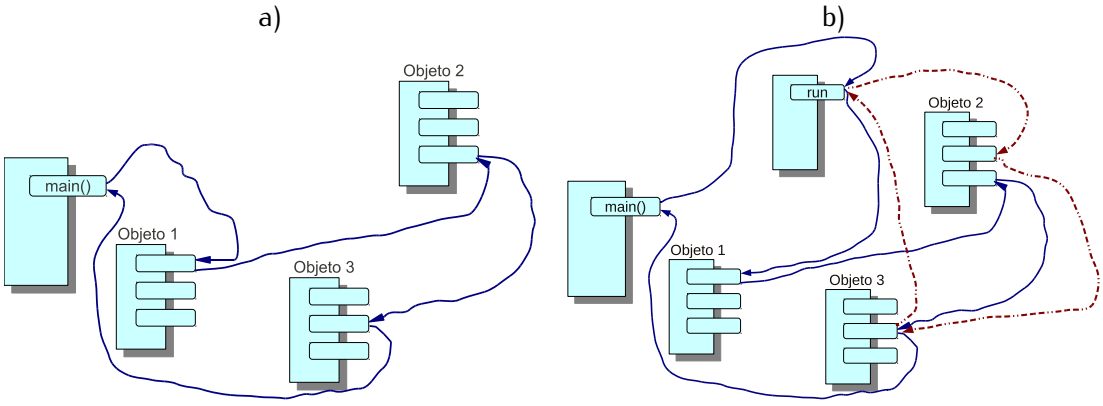
Concepto de hebra en Java

- La mayoría de las implementaciones de Java incluyen un compilador que genera código intermedio independiente de la máquina (llamado `bytecode`) más un intérprete de dicho código intermedio.
- El bytecode es interpretado por un proceso denominado **Máquina Virtual Java (JVM)**.
- Las hebras Java se ejecutan dentro de la JVM y comparten los recursos de este proceso del sistema operativo.
- Java ofrece una interfaz para manejar hebras.
- Una hebra Java es una instancia de una clase que implementa la interfaz `Runnable`, y/o instancias de una subclase de la clase `Thread`.
- Los métodos de las clases `Thread` y `Object` (definidas en el paquete `java.lang`) son los que hacen posible la gestión y sincronización de hebras en Java.

Ejecución de una hebra en Java

a) Por defecto, al ejecutar un programa Java, tenemos una hebra asociada al método `main()`, que va recorriendo distintos objetos conforme se invocan los correspondientes métodos.

b) Dentro de la hebra `main`, se pueden crear varias hebras que pueden ejecutar métodos del mismo o de diferentes objetos en el mismo o en diferente momento.



8.2 Creación de hebras en Java

Creación de hebras en Java

Para ejecutar una hebra en Java es necesario definir (al menos) una clase *C* que contenga un método **`run`**. Cada hebra ejecuta el código de dicho método para una instancia de *C*. Esto se puede implementar de dos formas:

- *C* es una clase derivada de la clase **`Thread`**:
  - **`Thread`** es una clase predefinida en el lenguaje.
  - Se impide que *C* sea derivada de otra clase distinta de **`Thread`**.
- *C* es una clase que implementa la interfaz **`Runnable`**:
  - **`Runnable`** es un interfaz predefinido en el lenguaje.
  - *C* puede extender (ser derivada de) una clase base cualquiera.
  - Se requiere adicionalmente la creación de una instancia de **`Thread`** por cada hebra a ejecutar. Esta instancia guarda una referencia a la instancia de *C*.

Nosotros usaremos la segunda opción precisamente por su mayor flexibilidad, a pesar de que es ligeramente más complejo. Habrá tantas clases *C* como sea necesario.

Uso de `Runnable` y `Thread`

Por cada hebra a ejecutar es necesario crear, además de una instancia *r* de *C*, una instancia *t* de **`Thread`** (*t* contiene una referencia a *r*). Hay dos posibilidades:

- El objeto *r* y después *t* se crean de forma independiente en la aplicación como dos objetos distintos.
- La clase *C* incluye como variable de instancia el objeto *t* (el objeto *r* tiene una referencia a *t*, o bien decimos que el objeto **`Runnable`** *encapsula* al objeto **`Thread`**).

nosotros usaremos la segunda opción ya que fuera del código de *C* solo tendremos que gestionar un objeto *C* en lugar de dos de ellos (uno *C* y otro **`Thread`**).

Creación y ejecución de una hebra.

Definiremos una clase **`TipoHebra`**, la cual

- puede ser derivada de una clase base **`Base`** cualquiera (no es necesario).
- tiene una variable de instancia pública de nombre, por ejemplo, **`thr`**, de la clase **`Thread`**.
- en su constructor crea el objeto **`thr`**.

Para ejecutar una de estas hebras es necesario:

1. Declarar una instancia **`obj`** de la clase **`TipoHebra`** (internamente se crea **`obj.thr`**).
2. Usar el método **`start`** de **`Thread`** sobre **`obj.thr`**, es decir, ejecutar **`obj.thr.start()`**.

La nueva hebra comienza la ejecución concurrente de **`obj.run()`**.

Ejemplo de una hebra

La definición de **TipoHebra** puede, a modo de ejemplo, ser así:

```
1  class TipoHebra implements Runnable // opcionalmente: extends ....
2  {
3    long siesta  ;       // tiempo que duerme la hebra
4    public Thread thr ; // objeto hebra encapsulado
5
6    public TipoHebra( String nombre, long siesta )
7    { this.siesta = siesta;
8      thr = new Thread( this, nombre );
9    }
10   public void run()
11   { try
12     { while ( true )
13       { System.out.println( "Hola, soy "+thr.getName() );
14         if ( siesta > 0 ) Thread.sleep( siesta );
15       }
16     }
17     catch ( InterruptedException e )
18     { System.out.println( "me fastidiaron la siesta!" );
19     }
20   }
21 }
```

Programa principal

Lanza una hebra y después hace **join** (la espera) y **getName**.

```
23 class Principal1
24 {
25   public static void main( String[] args ) throws InterruptedException
26   {
27     if ( args.length < 1 )
28     {  System.out.println( "Error: falta valor de 'siesta'" );
29        System.exit(1);
30     }
31     long siesta = Long.parseLong( args[0] ) ;
32     TipoHebra obj = new TipoHebra("hebra 'obj'", siesta); // crear hebra
33
34     obj.thr.start();      // lanzar hebra
35     Thread.sleep( 100 ); // la hebra principal duerme 1/10 sec.
36     obj.thr.join();       // esperar a que termine la hebra
37   }
38 }
```

8.3 Estados de una hebra Java

Estados de una hebra Java

- **Nueva**: Cuando el objeto hebra es creado, por ejemplo con:

```
Thread thr = new Thread (a);
```

- **Ejecutable**: Cuando se invoca al método **start** de una nueva hebra:

```
thr.start ();
```

- **En ejecución**: Cuando la hebra está ejecutándose en alguna CPU del computador. Si una hebra en este estado ejecuta el método **yield**, deja la CPU y pasa el estado ejecutable:

```
Thread.yield ();   // metodo static de la clase Thread
```

- **Bloqueada**: Una hebra pasa a este estado:
  - cuando llama al método **sleep**, volviendo al estado ejecutable cuando retorne la llamada:

```
    Thread.sleep (tiempo_milisegundos}; // metodo estático
```

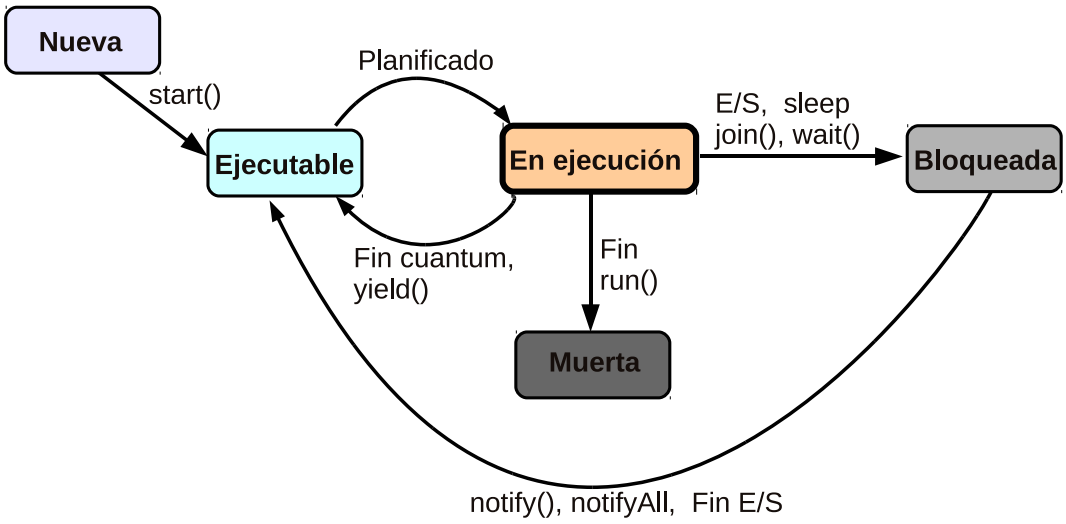
  - cuando llama a **wait** dentro de un bloque o un método sincronizado, volviendo al estado ejecutable cuando otra hebra llame a **notify** o a **notifyAll**
  - cuando llama a **join** para sincronizarse con la terminación de otra hebra que aún no ha terminado:

```
    thr.join ();
```

  - cuando ejecuta alguna operación de e/s bloqueante.
- **Muerta**: Cuando su método **run** termina.

Transiciones entre estados

El siguiente diagrama muestra de forma resumida las diferentes transiciones entre estados y los métodos que los provocan:



8.4 Prioridades y planificación.

Prioridad de una hebra

- Cada hebra tiene una prioridad que afecta a su planificación. Una hebra hereda la prioridad de la hebra que la creó. La prioridad es un valor entero entre **`Thread.MIN_PRIORITY`** y **`Thread.MAX_PRIORITY`** (dos constantes).
- La prioridad actual de una hebra puede obtenerse invocando el método **`getPriority`**, y se modifica con **`setPriority`**. En el ejemplo creamos una hebra y decrementamos su prioridad antes de llamar a **`start`**:

```
thr = new Thread( obj );
thr.setPriority( thr.getPriority()-1 );
thr.start();
```

Planificación de hebras

- El planificador de hebras de Java asegura que la hebra ejecutable con mayor prioridad (o una de ellas, si hay varias) pueda ejecutarse en la CPU. Si una hebra con mayor prioridad que la actual pasa a estar en estado ejecutable, la hebra actual pasa al estado ejecutable, y la hebra ejecutable de mayor prioridad es ejecutada (se apropia de la CPU).
- Si la hebra actual invoca a **`yield`**, se suspende, o se bloquea, el planificador elige para ejecución la próxima hebra de mayor prioridad.
- Otro aspecto de la planificación es el *tiempo compartido*, es decir la alternancia en el uso de la CPU por parte de las hebras ejecutables de mayor prioridad.

8.4.1 Un ejemplo más completo

Ejemplo de un vector de hebras.

El este ejemplo se crea un array de hebras de acuerdo con el esquema visto en la primera sección. Cada hebra es una instancia de **`Dormilona`**:

```
44 class Dormilona implements Runnable
45 {
46   int vueltas = 0 ; // número de veces que duerme (0 == infinitas)
47   int siesta  = 0 ; // tiempo máximo que duerme cada vez
48   public Thread thr ; // objeto hebra encapsulado
49
50   public Dormilona( String p_nombre, int p_vueltas, int p_siesta )
51   {
52     siesta  = p_siesta ;
53     vueltas = p_vueltas ;
54     thr     = new Thread( this, p_nombre ) ;
55   }
56
57   //...
```

Ejemplo de un vector de hebras (2)

El método **run** contiene el código que ejecutan todas las hebras:

```
58   //...
59   public void run()
60   { try
61     { Random random = new Random(); // crea generador de números aleatorios
62       // dormir un numero de veces igual a "vueltas"
63       for ( int i=0 ; i < vueltas || vueltas == 0 ; i++ )
64       { // imprimir nombre
65         System.out.println( "Vuelta no."+i+", de " +thr.getName());
66         // duerme un tiempo aleatorio entre 0 y siesta-1 milisegundos
67         if ( siesta > 0 )
68           Thread.sleep( random.nextInt( siesta ) );
69       }
70     }
71     catch( InterruptedException e )
72     { System.out.println( Thread.currentThread().getName()+
73                           ": me fastidiaron la siesta!");
74     }
75   } // fin run
76 } // fin de la clase Dormilona
```

Ejemplo de un vector de hebras (4)

El método main lee los parámetros.

```
79 class Principal2
80 {
81   public static void main( String[] args )
82   { try
83     { if ( args.length < 3 )
84       { System.out.println( "falta num_hebras, t_max_siesta, vueltas" );
85         System.exit( 1 );
86       }
87       int nHebras = Integer.parseInt( args[0] );
88       int siesta  = Integer.parseInt( args[1] );
89       int vueltas = Integer.parseInt( args[2] );
90
91       System.out.println( "nHebras = "+nHebras+", vueltas = "+vueltas+
92                           ", tsiesta = "+siesta );
93       //....
```

Ejemplo de un vector de hebras (5)

Finalmente, se crean las hebras en el vector **vhd** y se lanzan todas

```
94       //....
95
96       Dormilona[] vhd = new Dormilona[nHebras] ; // crea vector de hebras
```

```
97
98        for ( int i =0 ; i < nHebras ; i++ )
99        { String nombre = "Dormilona no."+i ;
100         vhd[i] = new Dormilona(nombre,vueltas,siesta); // crear hebra i.
101         vhd[i].thr.start();                            // comienza su ejec.
102       }
103       // la hebra principal duerme durante un segundo
104       Thread.sleep( 1000 );
105       System.out.println( "main(): he terminado de dormir" );
106
107       // esperar a que terminen todas las hebras creadas
108       for( int i = 0 ; i < nHebras ; i++ )
109         vhd[i].thr.join();
110     }
111     catch( InterruptedException e )
112     { System.out.println( "main(): me fastidiaron la siesta!" );
113     }
114   }
115 }
```

8.5 Interacción entre hebras Java. Objetos compartidos

Interacción entre hebras Java. Objetos compartidos

- En muchos casos, las hebras de un programa concurrente en Java han de interactuar para comunicarse y cooperar.
- La forma más simple en la que múltiples hebras Java pueden interactuar es mediante un objeto cuyos métodos pueden ser invocados por un conjunto de hebras. Invocando los métodos de este *objeto compartido* las hebras modifican el estado del mismo.
- Dos hebras se pueden comunicar si una escribe el estado del objeto compartido y otra lo lee. Asimismo, la cooperación entre hebras se puede llevar a cabo mediante la actualización de alguna información encapsulada en un objeto compartido.
- La ejecución concurrente entre hebras Java supone un entrelazamiento arbitrario de instrucciones atómicas que puede dar lugar a estados incorrectos de los objetos compartidos por varias hebras.

8.5.1 Implementando exclusión mutua en Java. Código y métodos sincronizados.

Exclusión mutua en Java. Código y métodos sincronizados.

- En Java, cda objeto tiene un cerrojo interno asociado. El cerrojo de un objeto solamente puede ser adquirido por una sola hebra en cada momento.
- El cualificador **`synchronized`** sirve para hacer que un bloque de código o un método sea protegido por el cerrojo del objeto.

- Para ejecutar un bloque o un método sincronizado, las hebras deben adquirir el cerrojo del objeto, debiendo esperar si el cerrojo ha sido ya adquirido por otra hebra.
- Si **obj** es una referencia a un objeto (de cualquier clase), la siguiente construcción hace que las hebras tengan que adquirir el cerrojo del objeto para ejecutar el bloque de código sincronizado.

```
synchronized( obj )
{
    // bloque de codigo sincronizado
}
```

Si todas las secciones críticas están en el código de un único objeto, podremos utilizar **this** para referenciar al objeto cuyo cerrojo vamos a utilizar:

```
synchronized( this )
{   // bloque de codigo sincronizado
}
```

Podemos hacer que el cuerpo entero de un método sea código sincronizado:

```
tipo metodo( ... )
{ synchronized( this )
  {   // codigo del metodo sincronizado
  }
}
```

La siguiente construcción es equivalente a la anterior:

```
synchronized tipo metodo( ... )
{   // codigo del metodo sincronizado
}
```

8.5.2 Ejemplo: Cálculo de múltiplos

Clase contador de número de múltiplos

Programa multihebra para calcular el número de múltiplos de 2 y 3 entre 1 y 1000 ambos incluidos.

- Una hebra contará múltiplos de 2 y otra de 3.
- Para llevar la cuenta se crea un único objeto compartido de la clase **Contador**, que encapsula un valor entero.
- La clase **Contador** incluye un método para incrementarlo y otro para obtener el valor.
- El uso compartido requiere usar esas métodos en exclusión mutua.

Clase contador de número de múltiplos

La clase **Contador** puede declararse así:

```
1  class Contador
2  { private long valor;
3    public Contador( long inicial )
4    { valor = inicial ;
5    }
6    void retrasoOcupado() // ocupa la CPU durante cierto tiempo
7    { long tmp = 0 ;
8      for( int i = 0 ; i < 100000 ; i++ )
9        tmp = tmp*i-tmp+i ;
10   }
11   public synchronized void incrementa () // incrementa valor en 1
12   { long aux = valor ; // hace copia local del valor actual
13     retrasoOcupado() ; // permite entrelazado cuando no se hace en EM
14     valor = aux+1 ;     // escribe el valor compartido (incrementado)
15   }
16   public synchronized long getvalor () // devuelve el valor actual
17   {  return valor;
18   }
19 }
```

Hebras que cuentan los múltiplos

Cada una de las hebras incrementa el contador cuando encuentra un múltiplo:

```
21 class Hebra implements Runnable
22 { int numero ;                // cuenta múltiplos de este número
23   public Thread thr ;         // objeto encapsulado
24   private Contador cont;      // contador de número de múltiplos
25
26   public Hebra( String nombre, int p_numero, Contador p_contador )
27   { numero = p_numero;
28     cont   = p_contador;
29     thr = new Thread( this, nombre );
30   }
31   public void run ()
32   { for ( int i = 1 ; i <= 1000 ; i++ )
33       if (i % numero == 0)
34         cont.incrementa();
35   }
36 }
```

¿ que pasaría si los dos métodos de **Contador** no fueran **synchronized** ?

Programa principal

Lanza dos hebras e imprime la cuenta calculada y la correcta:

```
38 class Multiplos
```

```
39 { public static void main( String[] args )
40   { try
41     { Contador contH = new Contador(0); // contador usado por la hebras
42       Hebra[] vc = new Hebra[2] ;
43       vc[0] = new Hebra( "Contador2 ", 2, contH);
44       vc[1] = new Hebra( "Contador3 ", 3, contH);
45       vc[0].thr.start(); vc[1].thr.start(); // lanza hebras
46       vc[0].thr.join();  vc[1].thr.join();  // espera terminación
47       System.out.println("Resultado hebras  : "+contH.getvalor());
48       long contV = 0 ;  // contador de verificacion
49       for ( int i = 1 ; i <= 1000 ; i++ )
50       { if ( i % 2 == 0 ) contV = contV + 1 ;
51         if ( i % 3 == 0 ) contV = contV + 1 ;
52       }
53       System.out.println("Resultado correcto: " + contV);
54     }
55     catch (InterruptedException e)
56     { System.out.println ("ha ocurrido una excepcion.");
57     }
58   }
59 }
```

8.6 Compilando y ejecutando programas Java

Compilar programas Java

Los programas fuente se encuentran en archivos de extensión `.java`. Cada uno de ellos contendrá una o mas definiciones de clases. Para compilar un fuente java (en un archivo `principal.java`, por ejemplo), hay que escribir:

```
javac principal.java
```

Si el programa `principal` hace mención a clases definidas en otros archivos, el compilador intentará compilarlos también, aunque se podría hacer manualmente antes:

```
javac otro1.java
javac otro2.java
javac ....
```

esto generará un archivo objeto (con *bytecode*) por cada clase definida en `principal.java` o en los otros archivos.

Ejecutar programas Java

Los archivos con *bytecode* tienen el nombre de la clase y la extensión `.class`.

Una de las clases compiladas debe contener el método principal (`main`), y lo usual (aunque no necesario) es que dicha clase se llame igual que el fuente que lo contiene (en este caso `principal`). Si esto ocurre, el programa se ejecuta con la orden:

```
java principal
```

para poder encontrar los archivos binarios `javac` y `java` hay que asignar la variable de entorno `PATH`, en las aulas:

```
export PATH=/fenix/depar/lsi/java/jdk1.5.0linux/bin:.:${PATH}
```

Chapter 9

Práctica 2. Programación de monitores con hebras Java.

9.1 Objetivos

Objetivos

- Conocer cómo construir monitores en Java, tanto usando la API para manejo de hebras Java, como usando un conjunto de clases (el paquete **`monitor`**) que permite programar monitores siguiendo la misma semántica de los monitores de *Hoare*.
- Conocer varios problemas sencillos de sincronización y su solución basada en monitores en el lenguaje Java:
  - Diseñar una solución al problema del *productor-consumidor con buffer acotado* basada en monitores e implementarla con un programa Java multihebra, usando el paquete **`monitor`**.
  - Diseñar una solución al problema de los *fumadores*, visto en la práctica 1, basada en monitores e implementarla con un programa Java multihebra, usando el paquete **`monitor`**.
  - Diseñar e implementar una solución al problema del *barbero durmiente* usando el paquete **`monitor`**.

9.2 Implementación de monitores nativos de Java

9.2.1 Monitores como Clases en Java

Monitores como Clases en Java

Para construir un monitor en Java, creamos una clase con sus métodos sincronizados. De esta forma solamente una hebra podrá ejecutar en cada momento un método del objeto.

El siguiente ejemplo muestra un monitor que implementa un contador más general que el visto en el seminario:

```
class Contador                    // monitor contador
{ private volatile int actual;
```

```
  public Contador( int inicial )
  { actual = inicial ;
  }
  public synchronized void inc()
  { actual++ ;
  }
  public synchronized void dec()
  { actual-- ;
  }
  public synchronized int valor()
  { return actual ;
  }
}
```

9.2.2 Métodos de espera y notificación.

Métodos de espera y notificación (1)

- En Java no existe el concepto de variable condición. Podríamos decir que cada monitor en Java tiene una única variable condición anónima.
- Los monitores basados en la biblioteca estándar de Java implementan una versión restringida de la semántica Señalar y Continuar (SC).
- Los mecanismos de espera y notificación se realizan con tres métodos de la clase **`Object`** (los tienen implícitamente todas las clases): **`wait`**, **`notify`** y **`notifyAll`**.
- Estos métodos solamente pueden ser llamados por una hebra cuando ésta posee el cerrojo del objeto, es decir, desde un bloque o un método sincronizado (protegido con **`synchronized`**).

Métodos de espera y notificación (2)

`wait()`
Provoca que la hebra actual se bloquee y sea colocada en una cola de espera asociada al objeto monitor. El cerrojo del objeto es liberado para que otras hebras puedan ejecutar métodos del objeto. Otros cerrojos poseídos por la hebra suspendida son retenidos por esta.

`notify()`
Provoca que, si hay alguna hebra bloqueada en **`wait`**, se escoja una cualquiera de forma arbitraria, y se saque de la cola de **`wait`** pasando esta al estado preparado. La hebra que invocó **`notify`** seguirá ejecutándose dentro del monitor.

`notifyAll()`
Produce el mismo resultado que una llamada a **`notify`** por cada hebra bloqueada en la cola de **`wait`**: todas las hebras bloqueadas pasan al estado preparado.

Métodos de espera y notificación (3)

La hebra señalada deberá adquirir el cerrojo del objeto para poder ejecutarse.

- Esto significará esperar al menos hasta que la hebra que invocó **`notify`** libere el cerrojo, bien por la ejecución de una llamada a **`wait`**, o bien por la salida del monitor.

La hebra señalada no tiene prioridad alguna para ejecutarse en el monitor.

- Podría ocurrir que, antes de que la hebra señalada pueda volver a ejecutarse, otra hebra adquiera el cerrojo del monitor.

Inconvenientes de los monitores nativos de Java

- **Cola de espera única**: como sólo hay una cola de espera por objeto, todas las hebras que esperan a que se den diferentes condiciones deben esperar en el mismo conjunto de espera. Esto permite que una llamada a **`notify`**`()` despierte a una hebra que espera una condición diferente a la que realmente se notificó, incluso aunque existan hebras esperando la condición que realmente se pretendía notificar. La solución para esta restricción suele consistir en:
  - Sustituir algunas o todas las llamadas a **`notify`** por llamadas a **`notifyAll`**.
  - Poner las llamadas a **`wait`** en un bucle de espera activa:

```
while ( not  condicion_logica_desbloqueo ) { wait() ; }
```

- **No hay reanudación inmediata de hebra señalada**: esto se debe a la semántica SC que se sigue y es la segunda razón del uso de incluir las llamadas a **`wait`** en bucles de espera activa.

Ejemplo: Productor-Consumidor con buffer limitado

Monitor que implementa un buffer acotado para múltiples productores y consumidores:

```
1  class Buffer
2  { private int numSlots = 0, cont = 0;
3    private double[] buffer = null;
4    public Buffer( int p_numSlots )
5    { numSlots = p_numSlots ;
6      buffer = new double[numSlots] ;
7    }
8    public synchronized void depositar( double valor ) throws InterruptedException
9    { while( cont == numSlots ) wait();
10     buffer[cont] = valor; cont++;
11     notifyAll();
12   }
13   public synchronized double extraer() throws InterruptedException
14   { double valor;
15     while( cont == 0 ) wait() ;
16     cont--; valor = buffer[cont] ;
17     notifyAll();
18     return valor;
```

```
19    }
20  }
```

Productor-Consumidor: hebras consumidoras

```
47  class Consumidor implements Runnable
48  { private Buffer bb ;
49    int veces;
50    int numC ;
51    Thread thr ;
52    public Consumidor( Buffer pbb, int pveces, int pnumC )
53    { bb    = pbb;
54      veces = pveces;
55      numC  = pnumC ;
56      thr   = new Thread(this,"consumidor "+numC);
57    }
58    public void run()
59    { try
60      { for( int i=0 ; i<veces ; i++ )
61        { double item = bb.extraer ();
62          System.out.println(thr.getName()+", consumiendo "+item);
63        }
64      }
65      catch( Exception e )
66      { System.err.println("Excepcion en main: " + e);
67      }
68    }
69  }
```

Productor-Consumidor: hebras productoras

```
22  class Productor implements Runnable
23  { private Buffer bb ;
24    int veces;
25    int numP ;
26    Thread thr ;
27    public Productor( Buffer pbb, int pveces, int pnumP )
28    { bb    = pbb;
29      veces = pveces;
30      numP  = pnumP ;
31      thr   = new Thread(this,"productor "+numP);
32    }
33    public void run()
34    { try
35      { double item = 100*numP ;
36        for( int i=0 ; i<veces ; i++ )
37        { System.out.println(thr.getName()+", produciendo " + item);
38          bb.depositar( item++ );
39        }
```

```
40      }
41      catch( Exception e )
42      { System.err.println("Excepcion en main: " + e);
43      }
44    }
45  }
```

9.3 Implementación en Java de monitores estilo *Hoare*

Implementación en Java de monitores estilo *Hoare*

- Se ha desarrollado una biblioteca de clases Java (paquete **`monitor`**) que soporta la semántica de monitores estilo *Hoare*.
- Permite definir múltiples colas de condición y un **`signal`** supone la reactivación inmediata del proceso señalado (semántica *Señalar y espera Urgente, SU*).
- La documentación y el código de las clases están disponibles en:

  http://www.engr.mun.ca/ theo/Misc/monitors/monitors.html
- Para definir una clase monitor específica se debe definir una extensión de la clase **`AbstractMonitor`**.

Exclusión mutua y uso del paquete `monitor`

Para garantizar la exclusión mutua en el acceso a los métodos del monitor, se han de invocar los siguientes métodos de la clase **`AbstractMonitor`**:

- **`enter`**`()`: para entrar al monitor, se invoca al comienzo del cuerpo del método.
- **`leave`**`()`: para abandonar el monitor, se invoca al final del cuerpo (aunque cualquier **`return`** debe ir después).

Uso del paquete

- Es conveniente que el directorio **`monitor`**, conteniendo los archivos `.java` con las clases Java de dicho paquete, se encuentre colgando del mismo directorio donde se trabaje con los programas Java que usen el paquete **`monitor`**.
- En caso contrario, habría que redefinir la variable de entorno `CLASSPATH` incluyendo la ruta de dicho directorio.

Ejemplo: Clase Monitor para contar los días

Permite llevar un control de los días (considerados como grupos de 24 horas) dedicadas por todas las hebras de usuario.

```
3  class Contador_Dias extends AbstractMonitor
4  { private int num_horas = 0, num_dias = 0;
5    public void nueva_hora()
6    { enter() ;
7      num_horas++;
8      if ( num_horas == 24 )
9      { num_dias++;
10       num_horas=0;
11     }
12     leave() ;
13   }
14   public int obtener_dia( )
15   { enter() ;
16     int valor=num_dias;
17     leave() ;
18     return valor;
19   }
20 }
```

Objetos condición del paquete `monitor`

Es posible utilizar varias colas de condición, declarando diversos objetos de una clase denominada **`Condition`**.

- Para crear un objeto condición, se invoca el método **`makeCondition`**`()` de la clase **`AbstractMonitor`**. Ejemplo:

```
Condition puede_leer = makeCondition() ;
```

La clase **`Condition`** proporciona métodos de espera, notificación y para consultar el estado de la cola de condición:

- **`void await`**`()`: tiene la misma semántica que la primitiva **`wait`**`()` de los monitores estilo *Hoare*.
- **`void signal`**`()`: igual que la primitiva **`signal`**`()` de los monitores estilo *Hoare*.
- **`int count`**`()`: devuelve el número de hebras que esperan.
- **`boolean isEmpty`**`()`: indica si la cola de condición está vacía (**`true`**) o no (**`false`**).

Ejemplo: Monitor lectores-escritores (1)

Con prioridad a las lecturas.

```
4  class MonitorLE extends AbstractMonitor
5  {
6    private int num_lectores = 0 ;
7    private boolean escribiendo = false ;
8    private Condition lectura   = makeCondition();
9    private Condition escritura = makeCondition();
10
```

```
11    public void inicio_lectura()
12    { enter();
13      if (escribiendo) lectura.await();
14      num_lectores++;
15      lectura.signal();
16      leave();
17    }
18    public void fin_lectura()
19    { enter();
20      num_lectores--;
21      if (num_lectores==0) escritura.signal();
22      leave();
23    }
```

Ejemplo: Monitor lectores-escritores (2)

```
24    public void inicio_escritura()
25    { enter();
26      if (num_lectores>0 || escribiendo) escritura.await();
27      escribiendo=true;
28      leave();
29    }
30    public void fin_escritura() // prio. lect
31    { enter();
32      escribiendo=false;
33      if (lectura.isEmpty()) escritura.signal();
34      else lectura.signal();
35      leave();
36    }
37  } // fin clase monitor "Lect_Esc"
```

Probl. lectores-escritores: Hebra Lectora.

```
52  class Lector implements Runnable
53  {
54    private MonitorLE monitorLE ; // objeto monitor l.e. compartido
55    private int       nveces ; // numero de veces que lee
56    public Thread     thr    ; // objeto hebra encapsulado
57
58    public Lector( MonitorLE p_monitorLE, int p_nveces, String nombre )
59    { monitorLE  = p_monitorLE  ;
60      nveces     = p_nveces ;
61      thr        = new Thread(this,nombre);
62    }
63    public void run()
64    { for( int i = 0 ; i < nveces ; i++ )
65      { System.out.println( thr.getName()+": solicita lectura.");
66        monitorLE.inicio_lectura();
67          System.out.println( thr.getName()+": leyendo.");
```

```
68          aux.dormir_max( 1000 ) ;
69        monitorLE.fin_lectura();
70      }
71    }
72  }
```

Probl. lectores-escritores: Hebra Escritora.

```
74  class Escritor implements Runnable
75  {
76    private MonitorLE monitorLE ; // objeto monitor l.e. compartido
77    private int       nveces ; // numero de veces que lee
78    public Thread     thr    ;  // objeto hebra encapsulado
79
80    public Escritor( MonitorLE p_monitorLE, int p_nveces, String nombre )
81    { monitorLE  = p_monitorLE  ;
82      nveces     = p_nveces ;
83      thr        = new Thread(this,nombre);
84    }
85    public void run()
86    { for( int i = 0 ; i < nveces ; i++ )
87      { System.out.println( thr.getName()+": solicita escritura.");
88        monitorLE.inicio_escritura();
89          System.out.println( thr.getName()+": escribiendo.");
90          aux.dormir_max( 1000 );
91        monitorLE.fin_escritura ();
92      }
93    }
94  }
```

Probl. lectores-escritores: clase auxiliar

se ha usado el método (estático) **dormir_max** de la clase **aux**. Este método sirve para bloquear la hebra que lo llama durante un tiempo aleatorio entre 0 y el número máximo de milisegundos que se le pasa como parámetro. Se puede declarar como sigue:

```
39  class aux
40  {
41    static Random genAlea = new Random() ;
42    static void dormir_max( int milisecsMax )
43    { try
44      { Thread.sleep( genAlea.nextInt( milisecsMax ) ) ;
45      }
46      catch( InterruptedException e )
47      { System.err.println("sleep interumpido en 'aux.dormir_max()'");
48      }
49    }
50  }
```

9.4 Productor-Consumidor con buffer limitado

Ejercicio propuesto

Obtener una versión de la clase **Buffer**, que se desarrolló en una sección anterior para múltiples productores y consumidores, usando las clases vistas del paquete **monitor**.

Documentación para el portafolio

Los alumnos redactarán un documento donde se responda de forma razonada a cada uno de los siguientes puntos:

1. Describe los cambios que has realizado sobre la clase **Buffer** vista anteriormente, indicando qué objetos condición has usado y el propósito de cada uno.
2. Incluye el código fuente completo de la solución adoptada.
3. Incluye un listado de la salida del programa (para 2 productores, 2 consumidores, un buffer de tamaño 3 y 5 iteraciones por hebra.

9.5 El problema de los fumadores

El problema de los fumadores

En este ejercicio consideraremos de nuevo el mismo problema de los fumadores y el estanquero que ya vimos en la práctica 1:

- Se mantienen exactamente igual todas las condiciones de sincronización entre las distintas hebras involucradas.
- Se escribirá una clase hebra **Estanquero** y otra **Fumador**. De esta última habrá tres instancias, cada una almacenará el número de ingrediente que necesita (o lo que es equivalente: el número de fumador), que se proporcionará en el constructor.
- La interacción entre los fumadores y el estanquero será resuelta mediante un monitor **Estanco** basado en el paquete **monitor**.

A continuación se incluyen las plantillas de código que deben usarse para este problema.

Plantilla del monitor `Estanco`

Las sentencias de sincronización se encapsulan en el código del monitor, que tendrá esta declaración:

```
class Estanco extends AbstractMonitor
{ ...
  // invocado por cada fumador, indicando su ingrediente o numero
  public void obtenerIngrediente( int miIngrediente )
  {  ...
  }
  // invocado por el estanquero, indicando el ingrediente que pone
  public void ponerIngrediente( int ingrediente )
```

```
  {  ...
  }
  // invocado por el estanquero
  public void esperarRecogidaIngrediente()
  { ...
  }
}
```

Plantilla de la hebra `Fumador`

Cada instancia de la hebra de fumador guarda su número de fumador (el número del ingrediente que necesita):

```
class Fumador implements Runnable
{
  int miIngrediente;
  public Thread thr ;
  ...
  public Fumador( int p_miIngrediente, ... )
  {  ...
  }
  public void run()
  {
    while ( true )
    { estanco.obtenerIngrediente( miIngrediente );
      aux.dormir_max( 2000 );
    }
  }
}
```

Plantilla de la hebra `Estanquero`

El estanquero continuamente produce ingredientes y espera a que se recojan:

```
class Estanquero implements Runnable
{ public Thread thr ;
  ...
  public void run()
  { int ingrediente ;
    while (true)
    {
      ingrediente = (int) (Math.random () * 3.0); // 0,1 o 2
      estanco.ponerIngrediente( ingrediente );
      estanco.esperarRecogidaIngrediente() ;
    }
  }
}
```

Documentación para el portafolio

Los alumnos redactarán un documento donde se responda de forma razonada a cada uno de los siguientes puntos:

1. Describe qué objetos condición has usado y el propósito de cada uno.
2. Incluye el código fuente completo de la solución adoptada.
3. Incluye un listado de la salida del programa.

9.6 El problema del barbero durmiente.

Barbería: tipos de hebras.

El problema del barbero durmiente es representativo de cierto tipo de problemas reales: ilustra perfectamente la relación de tipo cliente-servidor que a menudo aparece entre los procesos.

- El problema trata sobre una barbería en la cual hay dos tipos de actores o hebras ejecutándose concurrentemente:
  - Una única hebra llamada *barbero*
  - Varias hebras llamadas *clientes* (un número fijo).
- En la barbería hay:
  - una única silla de cortar el pelo, y
  - una sala de espera para los clientes, con una cantidad de sillas al menos igual al número de clientes
- Las hebras no consumen CPU en estos casos:
  - Barbero: cuando no hay clientes que atender (decimos que *duerme*), cuando le está cortando el pelo a un cliente.
  - Cliente: cuando está en la sala de espera, cuando el barbero le está cortando el pelo, cuando está fuera de la barbería.

Barbería: Requerimientos de sincronización.

La sincronización entre hebras viene determinada por estas condiciones:

- El **barbero** ejecuta un bucle infinito, en cada iteración:
  - Si no hay clientes en la sala de espera, espera dormido a que llege un cliente a la barbería, o bien, si hay clientes en dicha sala, llama a uno de ellos.
  - Pela al cliente durante un intervalo de tiempo, cuya duración exacta la determina el barbero.
  - Avisa al cliente de que ha terminado de pelarlo.
- Cada **cliente** ejecuta un bucle infinito, en cada iteración:

- Entra a la barbería. Si el barbero está ocupado pelando, esperará en la sala de espera hasta que el barbero lo llame, o bien, si el barbero está dormido, el cliente despierta al barbero.
- Espera en la silla de corte a que el barbero lo pele, hasta que el barbero le avisa de que ha terminado. Sale de la barbería.
- Espera fuera de la barbería durante un intervalo de tiempo, cuya duración exacta la determina el cliente.

Ejercicio propuesto

Escribir un programa Java con hebras para el problema del barbero durmiente. La barbería se implementa usando un **monitor** para la sincronización entre la hebra del barbero y las hebras de clientes. Dicho monitor tiene tres procedimientos exportados, que se describen aquí:

- Los clientes llaman a **`cortarPelo`** para obtener servicio del barbero, despertándolo o esperando a que termine con el cliente anterior.
- El barbero llama a **`siguienteCliente`** para esperar la llegada de un nuevo cliente y servirlo.
- Cuando el barbero termina de pelar al cliente actual llama a **`finCliente`**, indicándole que puede salir de la barbería y esperando a que lo haga para pasar al siguiente cliente.

Usar las plantillas que se incluyen abajo.

Documentación para el portafolio

Los alumnos redactarán un documento con los siguientes elementos:

1. Descripción de los objetos condición usados y su propósito.
2. Código fuente completo de la solución.
3. Listado de la salida del programa.

Plantilla del Monitor `Barberia`

```
class Barberia extends AbstractMonitor
{  ...
  // invcado por los clientes para cortarse el pelo
  public void cortarPelo ()
  {  ...
  }
  // invocado por el barbero para esperar (si procede) a un nuevo cliente y sentarlo para el corte
  public void siguienteCliente ()
  {  ...
  }
  // invocado por el barbero para indicar que ha terminado de cortar el pelo
  public void finCliente ()
  {  ...
  }
}
```

Plantilla de las hebras `Cliente` **y** `Barbero`

```
class Cliente implements Runnable
{ public Thread thr ;
  ...
  public void run ()
  { while (true) {
      barberia.cortarPelo (); // el cliente espera (si procede) y se corta el pelo
      aux.dormir_max( 2000 );   // el cliente está fuera de la barberia un tiempo
    }
  }
}
class Barbero implements Runnable
{ public Thread thr ;
  ...
  public void run ()
  { while (true) {
      barberia.siguienteCliente ();
      aux.dormir_max( 2500 ); // el barbero está cortando el pelo
      barberia.finCliente ();
    }
  }
}
```

Chapter 10

Seminario 3. Introducción al paso de mensajes con MPI.

Introducción

- El objetivo de esta práctica es familiarizar al alumno con el uso de la interfaz de paso de mensajes MPI y la implementación OpenMPI de esta interfaz.
- Se indicarán los pasos necesarios para compilar y ejecutar programas usando OpenMPI.
- Se presentarán las principales características de MPI y algunas funciones básicas de comunicación entre procesos.

Enlaces para acceder a información complementaria

- Web oficial de OpenMPI.
- Instalación de OpenMPI en Linux.
- Ayuda para las funciones de MPI.
- Tutorial de MPI.

10.1 Message Passing Interface (MPI)

¿Qué es MPI?

- MPI es un estándar de programación paralela mediante paso de mensajes que permite crear programas portables y eficientes.
- Proporciona un conjunto de funciones que pueden ser utilizadas en programas escritos en C, C++, Fortran y Ada.
- MPI-2 contiene más de 150 funciones para paso de mensajes y operaciones complementarias (con numerosos parámetros y variantes).
- Muchos programas paralelos se pueden construir usando un conjunto reducido de dichas funciones (hay 6 funciones básicas).

Modelo de Programación en MPI

- El esquema de funcionamiento implica un número fijo de procesos que se comunican mediante llamadas a funciones de envío y recepción de mensajes.
- Se sigue como modelo básico el estilo SPMD (Single Program Multiple Data), en el que todos los procesos ejecutan un mismo programa.
- También se permite seguir un modelo MPMD (Multiple Program Multiple Data), en el que cada proceso puede ejecutar un programa diferente.
- La creación e inicialización de procesos no está definida en el estándar, depende de la implementación. En OpenMPI sería algo así:
  - `$ mpirun -np 4 -machinefile maquinas prog1`
  - Comienza 4 copias del ejecutable `prog1`.
  - El archivo `maquinas` define la asignación de procesos a máquinas.

Aspectos de implementación

- Hay que hacer: `#include "mpi.h"`
  - Define constantes, tipos de datos y los prototipos de las funciones MPI.
- Las funciones devuelven un código de error:
  - **`MPI_SUCCESS`**: Ejecución correcta.
- **`MPI_Status`** es una estructura que se obtiene cada vez que se completa la recepción de un mensaje. Contiene 2 campos:
  - `status.MPI_SOURCE`: proceso fuente.

  - `status.MPI_TAG`: etiqueta del mensaje.

- **Constantes** para representar tipos de datos básicos de C/C++ (para los mensajes en MPI): **`MPI_CHAR`**, **`MPI_INT`**, **`MPI_LONG`**, **`MPI_UNSIGNED_CHAR`**, **`MPI_UNSIGNED`**, **`MPI_UNSIGNED_LONG`**, **`MPI_FLOAT`**, **`MPI_DOUBLE`**, **`MPI_LONG_DOUBLE`**, etc.

- **Comunicador**: es tanto un grupos de procesos como un contexto de comunicación. Todas las funciones de comunicación necesitan como argumento un comunicador.

10.2 Compilación y ejecución de programas MPI

Compilación y ejecución de programas en OpenMPI

OpenMPI es una implementación portable y de código abierto del estándar MPI-2, llevada a cabo por una serie de instituciones de ámbito tanto académico y científico como industrial.

OpenMPI ofrece varios scripts necesarios para trabajar con programas aumentados con llamadas a funciones de MPI. Los más importantes son estos dos:

- **`mpicxx`**: para compilar y enlazar programas C++ que hagan uso de MPI.
- **`mpirun`**: para ejecutar programas MPI.

El programa **`mpicxx`** puede utilizarse con las mismas opciones que el compilador de C/C++ usual, p.e.:

- `$ mpicxx -c ejemplo.c`
- `$ mpicxx -o ejemplo ejemplo.o`

Compilación y ejecución de programas MPI

La forma más usual de ejecutar un programa MPI es :

- `$ mpirun -np 4 ./ejemplo`
- El argumento `-np` sirve para indicar cuántos procesos ejecutarán el programa ejemplo. En este caso, se lanzarán cuatro procesos `ejemplo`.
- Como no se indica nada más, OpenMPI lanzará dichos procesos en la máquina local.

10.3 Funciones MPI básicas

Funciones MPI básicas

Hay 6 funciones básicas en MPI:

- **`MPI_Init`**: inicializa el entorno de ejecución de MPI.
- **`MPI_Finalize`**: finaliza el entorno de ejecución de MPI.

- **`MPI_Comm_size`**: determina el número de procesos de un comunicador.
- **`MPI_Comm_rank`**: determina el identificador del proceso en un comunicador.
- **`MPI_Send`**: operación básica para envío de un mensaje.
- **`MPI_Recv`**: operación básica para recepción de un mensaje.

Inicializar y finalizar un programa MPI

Se usan estas dos sentencias:

```
int MPI_Init( int *argc, char ***argv )
```

- Llamado antes de cualquier otra función MPI.
- Si se llama más de una vez durante la ejecución da un error.
- Los argumentos `argc, argv` son los argumentos de la línea de orden del programa.

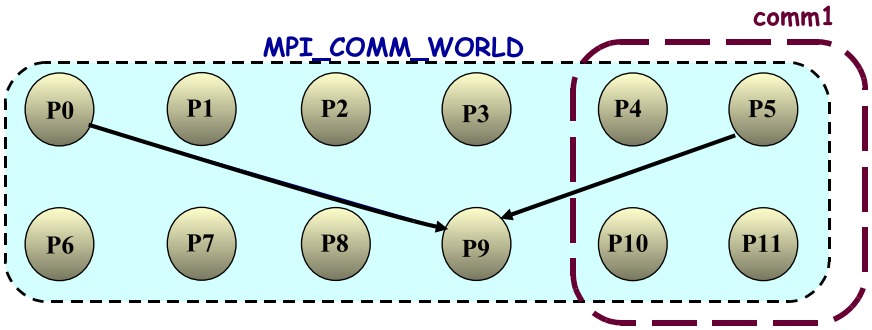
```
int MPI_Finalize( )
```

- Llamado al fin de la computación.
- Realiza tareas de limpieza para finalizar el entorno de ejecución

10.3.1 Introducción a los comunicadores

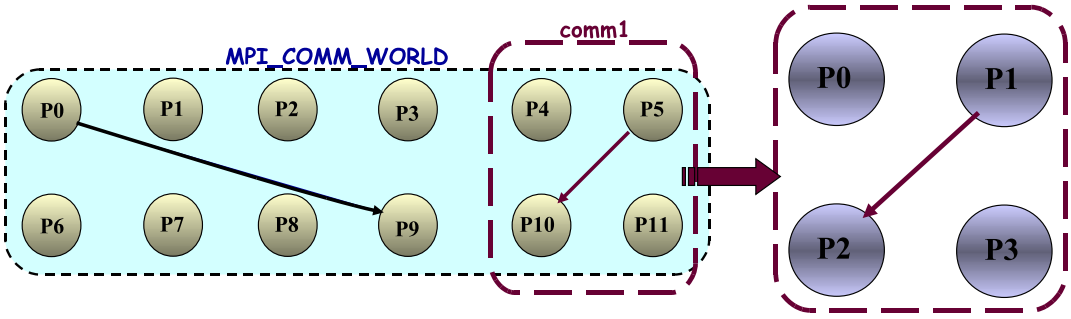
Introducción a los comunicadores MPI

- **Comunicador MPI**: es variable de tipo **`MPI_Comm`**.
- Un comunicador está constituido por:
  - Grupo de procesos: Subconjunto de procesos (pueden ser todos).
  - Contexto de comunicación: Ámbito de paso de mensajes en el que se comunican dichos procesos. Un mensaje enviado en un contexto sólo puede ser recibido en dicho contexto.
- Todas las funciones de comunicación de MPI necesitan como argumento un comunicador.



Introducción a los comunicadores

- La constante **MPI_COMM_WORLD** hace referencia al comunicador universal, un comunicador predefinido por MPI que incluye a todos los procesos de la aplicación (es el comunicador por defecto).
- La identificación de los procesos participantes en un comunicador es unívoca:
- Un proceso puede pertenecer a diferentes comunicadores.
- Cada proceso tiene un identificador: desde 0 a $P-1$ (P es el número de procesos del comunicador).
- Mensajes destinados a diferentes contextos de comunicación no interfieren entre sí.



Funciones para obtener información

```
int MPI_Comm_size( MPI_Comm comm, int *size )
```

- `size` : número de procesos que pertenecen al comunicador `comm`.
- ejemplo: **MPI_Comm_size**(**MPI_COMM_WORLD**,`&size`).

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

- `rank`: Identificador del proceso llamador en `comm`.

```
#include "mpi.h"
#include <iostream>
using namespace std ;
int main(int argc, char *argv[])
{ int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  cout << "Hola desde proc. "
       << rank << " de " << size << endl;
  MPI_Finalize();
  return 0;
}
```

```
$ mpicxx -o hola hola.cpp
$ mpirun -np 4 hola

Hola desde proc. 0 de 4
Hola desde proc. 2 de 4
Hola desde proc. 1 de 4
Hola desde proc. 3 de 4
```

10.3.2 Funciones básicas de envío y recepción de mensajes

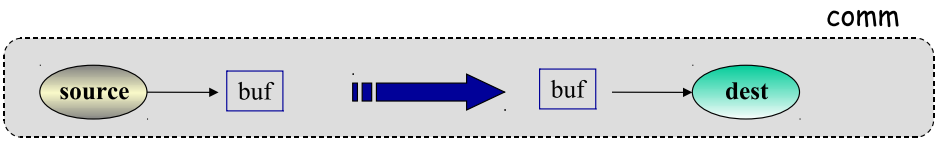
Envío y recepción de mensajes

```
int MPI_Send( void *buf, int num, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm )
```

- Envía los datos `num` elementos de tipo `datatype` almacenados a partir de `buf`) al proceso `dest` con la etiqueta `tag` (entero >= 0) dentro del comunicador `comm`.
- Implementa envio asíncrono seguro.

```
int MPI_Recv( void *buf, int num, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Status *status )
```

- Recibe mensaje de proceso `source` dentro del comunicador `comm` (recepción segura) y lo almacena en posiciones contiguas desde `buf`.
- Solo se recibe desde `source` con etiqueta `tag`, aunque existen argumentos comodín: **`MPI_ANY_SOURCE`**, **`MPI_ANY_TAG`**.



Envío y recepción de mensajes (2)

Los argumentos `num` y `datatype` determinan la longitud en bytes del mensaje. El objeto `status` es una estructura:

- Permite conocer el emisor (campo **`MPI_SOURCE`**), la etiqueta (campo **`MPI_TAG`**) y el número de items de un mensaje recibido.
- Para obtener la cuenta, el receptor debe conocer y proporcionar el tipo de los mismos. Se hace con **`MPI_Get_Count`**:

```
int MPI_Get_count( MPI_Status *status, MPI_Datatype dtype, int *num );
```

Ejemplo: Programa para dos procesos

```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if ( rank == 0 )
{   value = 100 ;
    MPI_Send( &value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD );
} else
    MPI_Recv( &value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
MPI_Finalize( );
```

Emparejamiento de operaciones de envío y recepción

En MPI, una operación de envío (con etiqueta e) desde un proceso A **encajará** con una operación de recepción en un proceso B solo si se cumplen cada una de estas tres condiciones:

- A nombra a B como receptor y e como etiqueta.
- B especifica **MPI_ANY_SOURCE**, o bien nombra explictamente a A como emisor
- B especifica **MPI_ANY_TAG**, o bien nombra explicitamente e como etiqueta

Si una operación encaja con varias (un envío con varias posibles recepciones, o una recepción con varios posibles envíos), entonces:

- Se seleccionará de entre esas varias **la primera en iniciarse** (esto facilita al programador garantizar propiedades de equidad).

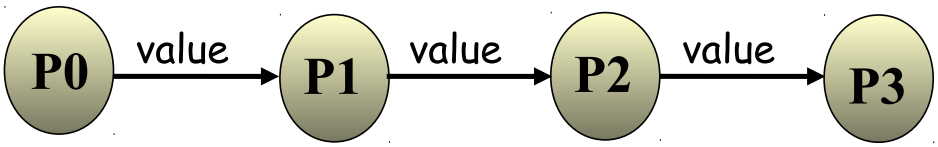
Interpretación de bytes transferidos

Es importante tener en cuenta que para determinar el emparejamiento MPI **no tiene en cuenta el tipo de datos ni la cuenta de items**. Es responsabilidad del programador asegurarse de que, en el lado del receptor:

- **Los bytes transferidos se interpretan con el mismo tipo de datos** que el emisor usó en el envío (de otra forma los valores leídos son indeterminados).
- **Se sabe exactamente cuantos items de datos** se han recibido (en otro caso el receptor podría leer valores indeterminados de zonas de memoria no escritas por MPI).
- **Se ha reservado memoria suficiente** para recibir todos los datos (de no hacerse, MPI escribiría erróneamente fuera de la memoria correspondiente a la variable especificada en el receptor).

Ejemplo: Difusión de mensaje en una cadena de procesos

```
int main(int argc, char *argv[])
{ int rank, size,value; MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);MPI_Comm_size(MPI_COMM_WORLD,&size );
  do
  { if (rank == 0)
    { scanf( "%d", &value );
      MPI_Send( &value, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD );
    }
    else
    { MPI_Recv( &value, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, &status );
      if (rank < size-1)
        MPI_Send( &value, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD );
    }
    cout<< "Soy el proceso "<<rank<<" y he recibido "<<value<<endl;
  }
  while ( value >= 0 );
  MPI_Finalize(); return 0;
}
```



10.4 Paso de mensajes síncrono en MPI

Envío en modo síncrono

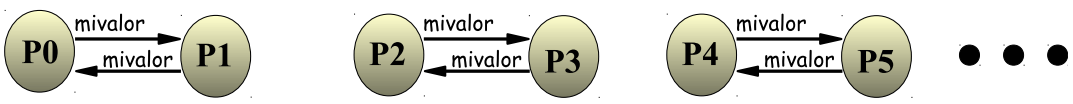
En MPI existe una función de envío **síncrono** (siempre es **seguro**):

```
int MPI_Ssend( void* buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm );
```

- Presenta los mismos argumentos que **MPI_Send**
- La operación de envío finalizará solo cuando el correspondiente **MPI_Recv** sea invocado y el receptor haya comenzado a recibir el mensaje, y además los datos hayan terminado de leerse en el emisor.
- Por **MPI_Ssend** es **seguro**: cuando devuelve el control, la zona de memoria que alberga el dato podrá ser reutilizada.
- Si la correspondiente operación de recepción usada es **MPI_Recv**, la semántica del paso de mensajes es puramente síncrona (existe una cita entre emisor y receptor).

Ejemplo: Intercambio síncrono entre pares de procesos

```
int main(int argc, char *argv[]) {
 int rank, size, mivalor, valor;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size( MPI_COMM_WORLD, &size );
 mivalor=rank*(rank+1);
 if (rank %2 == 0) {    // El orden de las operaciones es importante
     MPI_Ssend( &mivalor, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD );
     MPI_Recv( &valor, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD, &status );}
 else {
     MPI_Recv( &valor, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, &status );
     MPI_Ssend( &mivalor, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD );}
 cout<< "Soy el proceso "<<rank<<" y he recibido "<<valor<<endl;
 MPI_Finalize(); return 0;
}
```



10.5 Comunicación insegura

Comunicación insegura

- Las operaciones de comunicación vistas anteriormente son *seguras* (y por tanto pueden hacer esperar al proceso)
  - Incluso **MPI_Send** puede causar el bloqueo del emisor si no ha comenzado la operación de recepción correspondiente y no se dispone de memoria intermedia suficiente para copiar el mensaje completo.
- Se necesitan operaciones de comunicación no bloqueantes
  - Sondeo de mensajes:
    - **MPI_Iprobe**: Chequeo no bloqueante para un mensaje.
    - **MPI_Probe**: Chequeo bloqueante para un mensaje.
  - Envío-Recepción inseguro:
    - **MPI_Isend**: Inicia envío pero retorna antes de copiar en buffer.
    - **MPI_Irecv**: Inicia recepción pero retorna antes de recibir.
    - **MPI_Test**: Chequea si la operación no bloqueante ha finalizado.
    - **MPI_Wait**: Bloquea hasta que acabe la operación no bloqueante.

Comunicación asíncrona

El acceso no estructurado a un recurso compartido requiere comprobar la existencia de mensajes sin recibirlos.

```
int MPI_Iprobe( int source, int tag, MPI_Comm comm,
                int *flag, MPI_Status *status );
```

- No bloquea. Si hay algún mensaje, no se recibe (se puede hacer después con **MPI_Recv**, p.ej.).
- Si `flag > 0`, eso indica que hay un mensaje pendiente que encaja con (`source`, `tag`, `comm`).
- El argumento `status` permite obtener más información sobre el mensaje pendiente de recepción.

```
int MPI_Probe( int source, int tag, MPI_Comm comm, MPI_Status *status );
```

- Retorna sólo cuando hay algún mensaje que encaje con los argumentos.
- Permite esperar la llegada de un mensaje sin conocer su procedencia, etiqueta o tamaño.

Sondeo continuo de varias fuentes desconocidas

```
int rank, size, flag, buf, src,tag;
MPI_Status status ;
...
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
```

```
if (rank == 0) // proceso 0 recibe de todos los demás
{ int contador = 0;
  while ( contador < 10*(size-1) )
  { MPI_Iprobe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD, &flag, &status);
    if ( flag > 0 )
    { MPI_Recv( &buf, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                                  MPI_COMM_WORLD, &status );
      src = status.MPI_SOURCE; tag = status.MPI_TAG ;
      cout << "mensaje de " << src << " con tag= " << tag << endl;
      contador++;
    }
  }
  cout << "total de mensajes recibidos: " << contador << endl;
}
else  // resto de procesos envían 10 items cada uno a proceso 0
  for( int i = 0 ; i < 10 ; i++ )
     MPI_Send( &buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD );
...
```

Recepción con tamaño y fuente desconocidos

Se espera la llegada de un mensaje, una vez que llega, y antes de recibirlo, se reserva justo la memoria suficiente para contener el mensaje:

```
int num, *buf, source ;
MPI_Status status ;

// bloqueo hasta que se detecte un mensaje:
MPI_Probe( MPI_ANY_SOURCE, 0,comm, &status ) ;

// averigua el tamaño y el proceso emisor del mensaje:
MPI_Get_count( status, MPI_INT, &num );
source = status.MPI_SOURCE ;

// reserva memoria para recibir el mensaje:
buf = malloc( num*sizeof(int) );

// se recibe el mensaje:
MPI_Recv( buf, num, MPI_INT, source, 0, comm, &status );
```

Operaciones inseguras

Se pueden usar estas dos funciones:

```
int MPI_Isend( void* buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Irecv( void* buf, int count, MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm, MPI_Request *request)
```

Los argumentos son similares a **MPI_Send** excepto:

- Argumento `request`: Identifica operación cuyo estado se pretende consultar o se espera que finalice.
- No incluye argumento `status`: Se puede obtener con otras 2 funciones de chequeo de estado.

Cuando ya no se va a usar una variable **MPI_Request**, se puede usar:

```
int MPI_Request_free( MPI_Request *request )
```

- Libera la memoria usada por la variable `request`.

Operaciones inseguras. Chequeo de estado

Para comprobar si una operación insegura ha finalizado o no, se usa:

```
int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status )
```

- Escribe en `flag`.
- Si `flag > 0` entonces la operación identificada ha finalizado, libera `request` e inicializa `status`.

Para esperar bloqueado hasta que termine una operación, se usa:

```
int MPI_Wait( MPI_Request *request, MPI_Status *status )
```

- Produce bloqueo hasta que la operación chequeada finaliza (es segura).

Hay que tener en cuenta que es posible conectar operaciones inseguras con sus contrapartes seguras.

Intercambio de mensajes con operaciones inseguras

```
int main(int argc, char *argv[])
{ int rank, size, vecino, valor_env, valor_rec;
  MPI_Status status;
  MPI_Request request_env,request_rec;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  // calcular valor a enviar y número de proceso vecino:
  valor_env = rank*(rank+1);
  if (rank %2 == 0) vecino = rank+1 ;
  else              vecino = rank-1 ;
  // envio y recepción simultáneos (en cualquier orden)
  MPI_Irecv( &valor_rec,1,MPI_INT,vecino,0,MPI_COMM_WORLD,&request_rec);
  MPI_Isend( &valor_env,1,MPI_INT,vecino,0,MPI_COMM_WORLD,&request_env);
  // ... aquí se puede hacer algo que no use valor_rec ni altere valor_env
  // bloqueo hasta que sea seguro:
  MPI_Wait( &request_env, &status );
  MPI_Wait( &request_rec, &status );
  // ya esta:
  cout << "Soy el proceso "<<rank<<" y he recibido "<<valor<<endl ;
  MPI_Finalize(); return 0;
}
```

Chapter 11

Práctica 3. Implementación de algoritmos distribuidos con MPI.

11.1 Objetivos

Objetivos

- El objetivo general es iniciarse en la programación de algoritmos distribuidos.
- Conocer varios problemas sencillos de sincronización y su solución distribuida mediante el uso de la interfaz de paso de mensajes MPI:
  - Diseñar una solución distribuida al problema del *productor-consumidor* con buffer acotado para varios productores y varios consumidores, usando MPI.
  - Diseñar diversas soluciones al problema de la *cena de los filósofos* usando MPI.

11.2 Productor-Consumidor con buffer acotado en MPI

11.2.1 Aproximación inicial en MPI

Aproximación inicial en MPI

- Supongamos que disponemos de una versión distribuida del problema del productor-consumidor que usa tres procesos y la interfaz de paso de mensajes MPI. Para ello, tendremos un proceso productor (proceso 0 del comunicador universal) que producirá datos, un proceso Buffer (proceso 1) que gestionará el intercambio de datos y un proceso consumidor que procesará los datos (proceso 2). El esquema de comunicación entre estos procesos se muestra a continuación:

- El proceso **Productor** se encarga de ir generando enteros comenzando por el 0 y enviárselos al proceso **Buffer**. El proceso **Consumidor** envía peticiones al proceso Buffer, recibe los enteros de **Buffer**, los imprime por pantalla y calcula su raíz cuadrada.
- El proceso **Buffer** debería atender las peticiones de ambos procesos (**Productor** y **Consumidor**).

Aproximación inicial. Código usando MPI

- Una aproximación inicial al problema se muestra en el siguiente código MPI para tres procesos, que modela la interacción de los mismos de una forma incorrecta al forzar una excesiva sincronización entre productor y consumidor.

```
#include "mpi.h"
...
#define Productor 0
#define Buffer  1
#define Consumidor 2
#define ITERS 20
...
int main( int argc, char *argv[] )
{ int rank, size ;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank ) ;
  MPI_Comm_size( MPI_COMM_WORLD, &size ) ;
  if ( rank == Productor ) productor() ;
  else if ( rank == Buffer ) buffer() ;
  else consumidor() ;
  MPI_Finalize( ) ;
  return 0;
}
```

Proceso Productor, Consumidor y Buffer

```
void productor()
{ for( unsigned int i = 0 ; i < ITERS; i++)
  { cout << "Productor produce valor " << i << endl ;
    MPI_Ssend( &i, 1, MPI_INT, Buffer, 0, MPI_COMM_WORLD );
  }
}
void consumidor()
{ int value, peticion=1; float raiz; MPI_Status status;
  for( unsigned int i = 0 ; i < ITERS ; i++ )
  { MPI_Ssend( &peticion, 1, MPI_INT, Buffer, 0, MPI_COMM_WORLD );
    MPI_Recv ( &value,    1, MPI_INT, Buffer, 0, MPI_COMM_WORLD, &status);
    cout << "Consumidor recibe valor "<<value<<" de Buffer "<<endl ;
    raiz = sqrt(value);
  }
}
void buffer()
```

```
{ int value,peticion;  MPI_Status status;
  for( unsigned int i = 0 ; i < ITERS ; i++ )
  { MPI_Recv(&value,    1, MPI_INT, Productor, 0, MPI_COMM_WORLD,&status);
    MPI_Recv(&peticion, 1, MPI_INT, Consumidor, 0, MPI_COMM_WORLD,&status);
    MPI_Ssend( &value,  1, MPI_INT, Consumidor, 0, MPI_COMM_WORLD);
    cout<< "Buffer envia valor " << value << " a Consumidor " << endl ;
  }
}
```

11.2.2 Solución con selección no determinista

Solución con selección no determinista

- Se debe permitir que el productor pueda enviar **TAM** datos sin tener que interrumpirse, y que el consumidor no se retrase cuando haya datos almacenados en el proceso buffer.
- Una forma de corregir dicho código consiste en usar una sentencia de **selección no determinista de órdenes con guarda** en el proceso **Buffer** que permita cierta asincronía entre productor y consumidor en función del tamaño del buffer temporal (**TAM**).
- En MPI, no hay ninguna sentencia de selección no determinista de órdenes con guarda, pero es fácil emularla con las funciones de sondeo **MPI_Probe** y/o **MPI_Iprobe**.

Proceso Buffer con selección no determinista

```
void buffer()
{ int value[TAM], peticion, pos=0,rama;
  MPI_Status status;
  for( unsigned int i = 0 ; i < ITERS*2 ; i++ )
  { if (pos==0) rama=0;          // el consumidor no puede consumir
    else if (pos==TAM) rama=1; // el productor no puede producir
    else                        // se puede consumir o producir
    { MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      if ( status.MPI_SOURCE == Productor ) rama = 0 ; else rama = 1 ;
    }
    switch(rama)
    { case 0 :
       MPI_Recv(&value[pos],1,MPI_INT, Productor,0,MPI_COMM_WORLD,&status);
       cout<< "Buffer recibe "<< value[pos] << " de Prod." << endl ;
       pos++; break;
      case 1 :
       MPI_Recv( &peticion,1,MPI_INT,Consumidor,0,MPI_COMM_WORLD,&status);
       MPI_Ssend( &value[pos-1],1,MPI_INT,Consumidor,0,MPI_COMM_WORLD);
       cout<< "Buffer envia "<< value[pos-1] << " a Cons." << endl ;
       pos--; break;
    }
 }
}
```

Ejercicio propuesto

Extender el programa MPI anteriormente presentado que implementa el productor-consumidor con buffer acotado (los fuentes del programa se proporcionan junto con el guión de prácticas) para que el proceso buffer dé servicio a 5 productores y 4 consumidores. Para ello, se lanzarán 10 procesos y asumiremos que los procesos $0 \ldots 4$ son productores, el proceso **`Buffer`** es el proceso 5 y el resto de procesos en el comunicador universal ($6 \ldots 9$) son consumidores.

Documentación para el portafolios

Los alumnos redactarán un documento donde se responda de forma razonada a cada uno de los siguientes puntos:

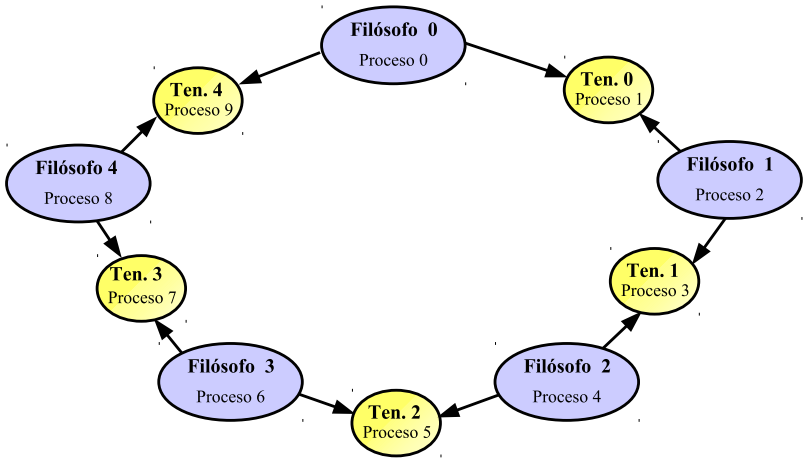
1. Describe qué cambios has realizado sobre el programa de partida y el propósito de dichos cambios.
2. Incluye el código fuente completo de la solución adoptada.
3. Incluye un listado parcial de la salida del programa.

11.3 Cena de los Filósofos

11.3.1 Cena de los filósofos en MPI

Cena de los filósofos en MPI.

- Se pretende realizar una implementación del problema de la cena de los filósofos en MPI utilizando el siguiente esquema:
- Tenemos 5 procesos filósofos y 5 procesos tenedor (10 procesos en total). Supondremos que los procs. filósofos se identifican con número pares y los tenedores con números impares. El filósofo i ($i = 0, ..., 4$) será el proc. $2i$ y el tenedor i será el $2i + 1$.



Cena de los filósofos. Programa principal

```
#include "mpi.h"
#include <iostream>
#include <time.h>
#include <stdlib.h>
  ...
void Filosofo(int id, int nprocesos); // Codigo proc. Filosofo
void Tenedor (int id, int nprocesos); // Codigo proc. Tenedor

int main( int argc, char** argv )
{ int rank,size;
  srand( time(0) );
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  if ( size != 10 )
  { if (rank == 0) cout << "El numero de procesos debe ser 10" << endl;
    MPI_Finalize( ); return 0;
  }
  if ( rank%2 == 0) Filosofo(rank,size); // los pares son filosofos
  else Tenedor(rank,size);               // los impares son tenedores
  MPI_Finalize();
  return 0;
}
```

Procesos filósofos

- En principio, cada filósofo realiza repetidamente la siguiente secuencia de acciones:
  - Pensar (sleep aleatorio).
  - Tomar los tenedores (primero el tenedor izquierdo y después el derecho).
  - Comer (sleep aleatorio).
  - Soltar tenedores (en el mismo orden).
- Las acciones pensar y comer pueden implementarse mediante un mensaje por pantalla seguido de un retardo durante un tiempo aleatorio. Las acciones de tomar tenedores y soltar tenedores pueden implementarse enviando mensajes de petición y de liberación a los procesos tenedor situados a ambos lados de cada filósofo.

Plantilla de la Función Filosofo

```
void Filosofo( int id, int nprocesos )
{
  int izq = (id+1)           % nprocesos ,
      der = (id+nprocesos-1) % nprocesos ;
  while ( true )
  { // solicita tenedor izquierdo
    cout << "Filosofo " << id << " solicita tenedor izq. " << izq << endl;
```

```
    // ...
    // solicita tenedor derecho
    cout << "Filosofo " << id << " coge tenedor der. " << der << endl;
    // ...
    cout << "Filosofo " << id << " COMIENDO" << endl ;
    sleep( (rand()%3)+1 );  // comiendo
    // suelta el tenedor izquierdo
    cout << "Filosofo "<<id<< " suelta tenedor izq. " << izq << endl;
    // ...
    // suelta el tenedor derecho
    cout << "Filosofo " << id << " suelta tenedor der. " << der << endl;
    // ...
    cout << "Filosofo " << id << " PENSANDO" << endl;
    sleep( (rand()%3)+1 ); // pensando
 }
}
```

Procesos tenedor

- Un tenedor solamente podrá ser asignado a uno de los dos filósofos que realicen la petición. Hasta que el tenedor no sea liberado no podrá ser asignado de nuevo. Cada proceso tenedor tendrá que ejecutar repetidamente la siguiente secuencia de acciones:
  - Esperar mensajes de petición de tenedor y recibir uno.
  - Esperar mensaje de liberación.

```
void Tenedor( int id, int nprocesos )
{
  int buf, Filo ;
  MPI_Status status;
  while ( true )
  { // Espera un peticion desde cualquier filosofo vecino ...
    // ...
    // Recibe la peticion del filosofo ...
    // ...
    cout << "Ten. " << id << " recibe petic. de " << Filo << endl;
    // Espera a que el filosofo suelte el tenedor...
    // ...
    cout << "Ten. " << id << " recibe liberac. de " << Filo << endl ;
  }
}
```

Ejercicio propuesto

- Implementar una solución distribuida al problema de los filósofos de acuerdo con el esquema descrito en las plantillas. Usar la operación síncrona de envío **MPI_Ssend** para realizar las peticiones y liberaciones de tenedores.

- El esquema propuesto (cada filósofo coge primero el tenedor de su izquierda y después el de la derecha) puede conducir a interbloqueo. Identificar la secuencia de peticiones de filósofos que conduce a interbloqueo en el programa y realizar pequeñas modificaciones en el programa (y en el comportamiento de las entidades que participan) que eliminan la posibilidad de interbloqueo (sin añadir nuevos procesos).

Documentación para el portafolios

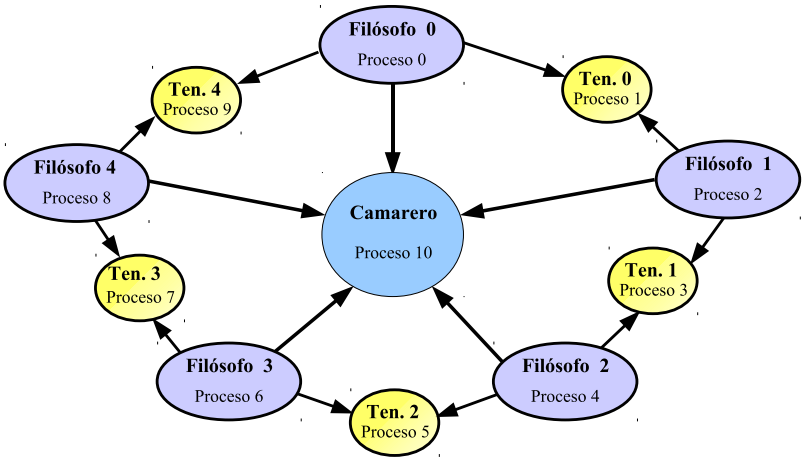
Los alumnos redactarán un documento donde se responda de forma razonada a cada uno de los siguientes puntos:

1. Describe los aspectos más destacados de tu solución al problema de los filósofos, la situación que conduce al interbloqueo y tu solución al problema del interbloqueo.
2. Incluye el código fuente completo de la solución adoptada para evitar la posibilidad de interbloqueo.
3. Incluye un listado parcial de la salida de este programa.

11.3.2 Cena de los filósofos con camarero en MPI

Cena de los filósofos con camarero

- Una forma de evitar la posibilidad de interbloqueo consiste en impedir que todos los filósofos intenten ejecutar la acción de "tomar tenedor" al mismo tiempo. Para ello podemos usar un proceso *camarero central* que permita sentarse a la mesa como máximo a 4 filósofos. Podemos suponer que tenemos 11 procesos y que el camarero es el proc. 10.



Proceso filósofo con Camarero central

- Ahora, cada filósofo ejecutará repetidamente la siguiente secuencia de acciones:
  - Pensar

  - Sentarse
  - Tomar tenedores
  - Comer
  - Soltar tenedores
  - Levantarse

- Cada filósofo pedirá permiso para sentarse o levantarse enviando un mensaje al camarero, el cual llevará una cuenta del número de filósofos que hay sentados a la mesa en cada momento.

Ejercicio propuesto

- Implementar una solución distribuida al problema de los filósofos con camarero central que se ha descrito, usando MPI.
- **Documentación para el portafolios**

  Los alumnos redactarán un documento donde se responda de forma razonada a cada uno de los siguientes puntos:

  1. Describe tu solución al problema de los filósofos con camarero central.
  2. Incluye el código fuente completo de la solución adoptada.
  3. Incluye un listado parcial de la salida del programa.