Sistemas Concurrentes y Distribuidos
Teoría y Práctica
Tomo II - Práctica

Manuel I. Capel Tuñón
Sandra Rodríguez Valenzuela

Contenidos

Práctica 1

Resolución de problemas de sincronización con hebras usando mecanismos de bajo nivel.

1.1 Resumen

El objetivo de esta práctica es aprender a utilizar la biblioteca `pthreads` POSIX 1003.1c [Gallmeister, 1995] para desarrollar programas concurrentes en C/C++. En lo que sigue, se supone que el alumno ya conoce las funciones de la interfaz `pthreads`, los mecanismos de sincronización (mutex y semáforos) y su utilización básica, parámetros, etc. Aquí se pretende dar los elementos de programación básicos para utilizar correctamente todo lo anterior. El texto que sigue pretende ser una ayuda para realizar correctamente la práctica y se ha estructurado en los siguientes apartados: (1) tutorial sobre manejo de *pthreads*, (2) problema del productor-consumidor y (3) problema de los fumadores.

1.2 Tutorial `pthreads`

Se trata de un programa que implementa un planificador de avisos, que se podría utilizar como una agenda de funcionalidad muy básica, con una interfaz de usuario muy simple que sólo admite peticiones de aviso en modo texto.

Funcionalidad del programa:

- se aceptarán continuamente, dentro de un bucle, peticiones del usuario;
- en cada una de dichas peticiones se introduce una línea completa de texto, hasta que se detecte un error o el fin de archivo en `stdin`;
- en cada línea, el primer símbolo que se puede formar es interpretado como el número de segundos que hay que esperar hasta mostrar el aviso;

	tipo de solución	núm. hebras/procesos	plantilla
1	síncrona	0	`p11.c`
2	asíncrona	> 2 procesos	`p12.c`
3	asíncrona	> 2 hebras	`p13.c`

Tabla 1.1: Partes del tutorial a realizar

- el resto de la línea (hasta 64 caracteres) es el propio mensaje de aviso.

Se comenzará el tutorial con una versión completa del programa totalmente *síncrona* (con la plantilla `p11.c`), es decir sin procesos ni hebras. Para después introducir hebras y obtener una versión*asíncrona* del programa, tal como se puede ver en la tabla resumen 1.1.

En el ejercicio 2 de la tabla 1.1 se han de crear múltiples procesos UNIX con `fork()`.

En el ejercicio 3 se han de crear múltiples hebras: 1 por cada aviso del usuario.

1.2.1 La versión síncrona del programa planificador de avisos

Consiste un programa secuencial síncrono que implementa un planificador de avisos con una única rutina `main()`. Un bucle procesa las peticiones del usuario hasta que `fgets` devuelve `NULL` (error o fin de archivo). Cada línea es explorada con `sscanf()` , para separar el número de segundos que ha de permanecer esperando el planificador (`%d`, *primera secuencia de dígitos*) de la cadena con el resto del mensaje –hasta 64 caracteres excluyendo `newline`–, que constituye el texto del mensaje del usuario: `%64[^\n`, *el resto de la línea.*

```
#include <errno.h>
#include <stdio.h>
FILE  *fp;
int main (int argc, char *argv[])
{
    int segundos;
    char linea[128];
    char mensaje[64];

   /* Se abre el fichero que contendra las salidas del programa */
   if ((fp = fopen("salidas", "w")) == NULL)
       fprintf(stderr, "Error en la apertura del fichero de salida\n"),exit(0);

    while (1) {
        printf ("Peticion de aviso> ");
        if (fgets (linea, sizeof (linea), stdin) == NULL) exit (0);
        if (strlen (linea) <= 1) continue;

        if (sscanf (linea, "%d %64[^\n]", &segundos, mensaje) < 2) {
            if (segundos==-1) break;
              else fprintf (stderr, "Entrada erronea\n");
        } else {
```

```
            sleep (segundos);
            fprintf (fp,"(%d) %s\n", segundos, mensaje);
        }
    }
  if (fclose(fp)) fprintf(stderr, "Error al cerrar el fichero\n");
}
```

Se utiliza el fichero `fp` para guardar la impresión de los avisos, así la salida del programa no interfiere con la entrada de peticiones. El programa termina si se le proporciona un número negativo de segundos (== -1); si no, demandará continuamente la entrada de un nuevo aviso del usuario.

1.2.2 Versión con múltiples procesos UNIX

Una forma de conseguir que las peticiones de aviso de los usuarios se procesen de forma asíncrona consiste en crear varios procesos UNIX: 1 por cada petición. Esto se puede hacer en UNIX con la función `fork()`, creando un nuevo proceso hijo por cada petición al *planificador* de avisos.

Uno de los inconvenientes de esta versión del programa consiste en que se han de "recoger" los procesos hijos que han recibido ya el aviso del planificador, si no el sistema se sobrecargaría porque el sistema mantendrá a los procesos hijos hasta que el programa completo termine.

Programando con C y UNIX, la manera normal de recoger a los procesos hijos que hayan terminado es llamar a una de las funciones `wait`, que de forma inmediata recoge a 1 proceso cada vez que se invoca, o bien, si no existe tal proceso en el momento de la llamada, la propia función terminaría, devolviendo como resultado el identificador `pid == 0`. La llamada `pid_t waitpid(pid_t pid, int *status, int options);` y el valor de `pid` que se pasa en el primer argumento puede ser:

- < -1: espera a cualquier proceso hijo cuyo identificador de grupo de procesos sea igual al valor absoluto de ese valor.
- $== -1$: espera a cualquier proceso hijo.
- $== 0$: espera a cualquier proceso hijo cuyo identificador de grupo de procesos coincida con el del proceso que llama a la función.
- > 0: espera al proceso hijo cuyo identificador coincide con ese valor.

El valor del argumento `options` se obtiene de hacer un OR con 0 o más de las siguientes constantes:

- `WNOHANG`: hacer volver la llamada inmediatamente si no ha acabado ningún proceso hijo.
- `WUNTRACED`: también vuelve si un proceso hijo se ha parado y no se le puede encontrar la pista con `ptrace(2)`.
- `WCONTINUED` (desde Linux 2.6.10): también vuelve si un proceso hijo parado se le ha hecho continuar mediante la entrega de la señal SIGCONT.

El bucle `do while()` siguiente puede servir de ejemplo para programar la llamada a `waitpid()` necesaria en el texto del proceso padre.

```
do { pid = waitpid ((pid_t)-1, NULL, WNOHANG);
     if (pid == (pid_t)-1)
        fprintf (stderr,"En la funcion wait"),exit(0);
   } while (pid != (pid_t)0);
```

1.2.3 Versión con múltiples hebras

En una aplicación en la que se necesitase crear un gran número de peticiones de aviso la solución anterior, basada en creación de procesos con `fork()`, no sería adecuada, ya que el sistema no nos dejaría crear muchos de procesos. Sin embargo, no existe ninguna limitación para crear cientos de hebras dentro de un programa. En la versión con múltiples hebras que presentamos a continuación, todas las hebras comparten el mismo espacio de direcciones, por lo que habrá que crear (utilizando memoria dinámica) una estructura de datos que contenga los valores del plazo de tiempo y del mensaje del aviso asociados a cada nueva petición del usuario.

```
typedef struct paquete_control {
    int         segundos;
    char        mensaje[64];
} paquete_control_t;
```

Se pasará un puntero a dicha estructura, como cuarto argumento de `pthread_create`, cuando se crea cada nueva hebra. Además, ya no hay necesidad de que la hebra `main()` recoja a las hebras terminadas, ya que se pueden crear con la opción `detached`, y de esta forma los recursos serán devueltos automáticamente al sistema cuando termine el programa. Se han de utilizar las siguientes funciones de la biblioteca pthread:

- `pthread_create()`
- `pthread_detach()`
- `pthread_self()`

La hebra se suspende durante el número de segundos especificado en su paquete de control y cuando pasa de nuevo a activa, imprime la cadena con el mensaje del usuario. `main()` ha de asignar memoria dinámica a la estructura `paquete_control`. Si la llamada a la función `sscanf` fallase cuando se explora la cadena de entrada, entonces la memoria dinámica asociada al paquete de control de la hebra debe ser desasignada (orden `free(...)` en la plantilla).

```
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
typedef struct paquete_control {
    int         segundos;
    char        mensaje[64];
```

```
} paquete_control_t;
FILE  *fp;
void *aviso (void *arg)
{
    /* Convertir el argumento de la funcion a puntero a un paquete de control*/
    /* Suspender el numero de segundos indicado en la peticion del usuario.*/
    /* Imprimir los segundos, el mensaje de usuario y el tiempo absoluto del  aviso.*/
    /* Liberar la memoria asignada al paquete de control */;

    return NULL;
}
int main (int argc, char *argv[])
{
    int estado;
    char linea[128];

    /* Declarar un puntero a un paquete de control para la peticion del usuario*/
    /* Declara aqui el resto de variables que te hagan falta*/

    if ((fp = fopen("salidas", "w")) == NULL) fprintf(stderr, "Error en la
          apertura del fichero de salida\n"),exit(0);
    while (1) {
        printf ("Peticion de aviso> ");
        if (fgets (linea, sizeof (linea), stdin) == NULL) exit (0);
        if (strlen (linea) <= 1) continue;
       /* Asignar memoria al puntero al paquete de control declarado*/;
       /* Utilizar "sscanf" para asignar los elementos de "paquete de control"*/
        if (sscanf(linea, "%d %64[^\n]", /** los segundos*/,
                    /** el mensaje*/)<2){
                if (/*los segundos==-1*/) break;
                else{
                     fprintf(stderr, "Entrada erronea\n");
                     free(/** la estructura que guarda los datos de la peticion*/);
                    }
         }
         else{
               /* Crear una hebra "aviso"*/
    }
   if (fclose(fp)) fprintf(stderr, "Error al cerrar el fichero\n");
}
```

Lista de actividades

Debes realizar las siguientes actividades en el orden indicado:

1. Completa las plantillas (`p11.c, p12.c, p13.c`), compila y prueba tus programas. Recuerda que los resultados de las ejecuciones se escribirán en el fichero "salidas" de tu directorio de trabajo.

2. Comprueba que los programas son correctos: verifica que los avisos aparecen en "salidas" en un orden consistente con los plazos de tiempo indicados cuando se crearon.

Documentación a entregar

Los alumnos redactarán un documento donde se responda de forma razonada y/o se incluya la información solicitada en los siguientes puntos:

1. Se describirá razonadamente en qué orden aparecen los avisos en el fichero "salidas" para la ejecución de los programas terminados correspondientes a las plantillas `p12.c, p13.c` y justificando dicho orden de aparición de los avisos.
2. Incluye el código fuente completo de las todas plantillas que hayas rellenado.
3. Un listado de la salida de los programas (para un número de avisos entre 20 y 40).

1.3 El problema del productor/consumidor

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. Aquí vamos a programar una solución de 2 hebras que utilizan un vector de ítems como almacenamiento intermedio de los datos producidos, antes de que los extraiga la hebra consumidora.

Para diseñar un programa que solucione este problema:

- El productor y el consumidor se programan como dos hebras independientes, ya que esto permite tener ocupadas las CPUs disponibles el máximo tiempo posible.
- Cada ítem producido ha de ser leído (ningún ítem se pierde).
- Ningún item se leerá más de una vez, lo cual implica:
  - el productor tendrá en ocasiones que esperar a escribir en el vector cuando haya creado un ítem pero el vector esté completamente ocupado por otros pendientes de ser leídos.
  - el consumidor debe esperar cuando vaya a leer del vector y dicho vector no contenga ningún ítem pendiente.
  - en algunas aplicaciones el orden de lectura debe coincidir con el de escritura, en otras podría ser irrelevante.

1.3.1 Plantillas de código

En esta práctica se diseñará e implementará un ejemplo sencillo en C/C++, de acuerdo con las siguientes simplificaciones:

- cada item de datos será un valor entero de tipo `int`,
- el orden en el que se leen los items es irrelevante,
- el productor produce los valores enteros en secuencia, empezando con el valor "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 datos

```
int producir_dato()
{
  static int contador = 1 ;
  return contador ++ ;
}
```

Para consumir:

```
void consumir_dato( int dato )
{
  cout << "dato recibido: " << dato << endl ;
}
```

Los subprogramas que ejecutan las hebras productora y consumidora sin incluir la sincronización entre las hebras para acceder al vector ni los accesos a dicho 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 ;
}
```

1.3.2 Actividades a realizar

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.
2. Diseña una solución, mediante semáforos, que permita realizar las esperas necesarias para cumplir el requisito anteriormente descrito.
3. Implementa la solución descrita en un programa C/C++ con hebras y semáforos POSIX, completando la plantilla incluida en este guión. Ten en cuenta que el programa debe escribir la palabra "fin" cuando hayan terminado las dos hebras.
4. Por último, comprueba que tu programa es correcto, es decir, verifica que cada dato producido es consumido exactamente una vez. Los datos aludidos pueden ser números enteros generados aleatoriamente.

1.4 El problema de los fumadores

Considerar un estanco en el que hay tres fumadores y un estanquero.

1. Cada fumador representa una hebra que realiza una actividad (fumar), invocando a una función `fumar()`, en un bucle infinito.
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.
3. El estanquero produce suministros para que los fumadores puedan fumar, también en un bucle infinito.
4. Para asegurar "concurrencia real" es importante tener en cuenta que la solución diseñada debe permitir que varios fumadores fumen simultáneamente.

Requisitos para que los fumadores puedan fumar y el funcionamiento del proceso estanquero

- Antes de fumar es necesario liar un cigarro, para ello 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 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 espera el aviso.

- El fumador desbloqueado toma los dos ingredientes del mostrador, avisa al estanquero para que pueda seguir sirviendo ingredientes y fuma durante un tiempo, después de liarse el cigarro.

- El estanquero vuelve a poner dos ingredientes aleatorios en el mostrador y se repite el ciclo.

Además para poder encontrar una solución segura al problema de los fumadores:

- Los fumadores retiran los ingredientes del mostrador de 1 sola vez.

- Un fumador sólo vuelve a liar cuando terminó de fumarse el último cigarrillo.

- "Fumar" lleva un tiempo, que se puede simular utilizando la función `sleep(n)`

- Los fumadores no pueden "reservarse" ingredientes que ya estén depositados en el mostrador para liar el siguiente cigarrillo. Por ejemplo: si el estanquero coloca tabaco y papel, mientras que el fumador que tiene las cerillas está fumando, el tabaco y el papel permanecerán en la mesa hasta que el fumador aludido termine de fumar su cigarrillo y retire de sola una vez dichos ingredientes del mostrador.

1.4.1 Actividades a realizar

Diseña e implementa una solución al problema en C/C++ usando cuatro hebras y los semáforos necesarios. La solución propuesta 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 suministros y qué suministros produce.
  - Cada fumador debe indicar cuándo espera, qué producto o productos espera, y cuándo comienza y finaliza de fumar.

Documentación a entregar

Los alumnos elegirán uno de los 2 ejercicios propuestos (*productor–consumidor* o *fumadores*) para su resolución. Redactarán un documento donde se responda de forma razonada a los siguientes puntos y/o se proporcione la información solicitada:

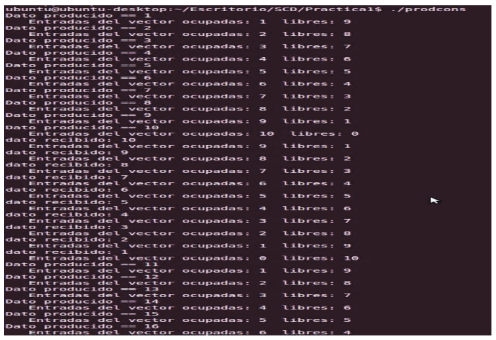
1. Indicar la variable o variables globales necesarias para programar una solución correcta al ejercicio elegido. Se ha de explicar la función de dichas variables dentro del programa presentado.

2. Describir los semáforos necesarios, la utilidad de cada uno de los mismos, el valor inicial y en qué puntos del programa se deben usar las operaciones `sem_wait()` y `sem_post()` y por qué.

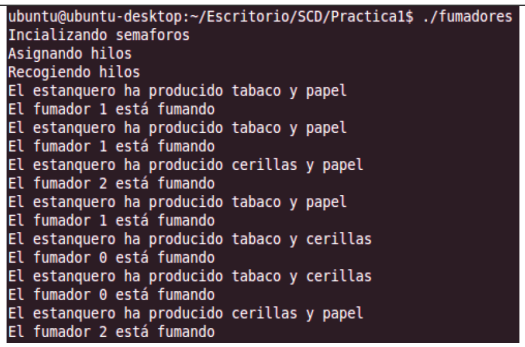
3. Incluye el código fuente completo de la solución adoptada.

4. Incluye un listado de la salida del programa (para un número de items entre 20 y 40).

5. Adicionalmente, un último punto en el que inluirás cualquier otra información que consideres relevante acerca del funcionamiento de tu programa, así como una discusión "no–formal" sobre su corrección.



Salida del productor–consumidor



Salida de fumadores

Práctica 2

Programación de monitores con hebras Java.

2.1 Resumen

El objetivo de esta práctica es familiarizarse con las diferentes formas de creación de hebras de Java. Este guión se corresponde con la primera parte de la práctica, que se continuará con varios ejercicios adicionales de programación de monitores y hebras, y propone una plantilla para completar que permite crear hebras en Java planificadas por tiempo compartido. Con la resolución de este primer ejercicio, también se pretende crear una infraestrucura común para resolver los siguientes. El texto que sigue pretende ser una ayuda para realizar correctamente la práctica y se ha estructurado en los siguientes apartados: introducción a la programación orientada a objetos, creación de hebras, planificación y asignación de prioridades, exclusión mutua y monitores en Java. Por último, se proponen varios ejercicios para programar.

2.2 Introducción

Los objetos y la concurrencia aparecen relacionados desde sus comienzos. De hecho, algunas de las versiones más tempranas de C++ incluyeron bibliotecas de clases para soportar la concurrencia [Lea, 2001].

> Concurrencia + Objetos = OOCP
> Object-oriented concurrent programming (OOCP) es un término que hace referencia a la computación que se realiza utilizando una colección de módulos pequeños, autocontenidos (denominados *objetos*) que ejecutan instrucciones e interaccionan concurrentemente a través de algún protocolo de comunicación unificado (denominado *paso de mensajes*). Este modelo de programación proporciona una buena base para descomponer los programas complejos y ejecutarlos eficientemente en multiprocesadores.

2.2.1 Programación orientada a objetos

Este método de programación ha sido reconocido en todo el mundo como el más útil para desarrollar software reutilizable y de alto nivel. La metodología que soporta la *programación orientada a objetos* (POO) es conceptualmente simple y lo suficientemente general para ser aplicada extensivamente en desarrollo de software. Se basa en 2 conceptos fundamentales: *objetos* que son unas entidades que representan *conocimiento*, esto es, datos y servicios; y un protocolo unificado de comunicación/activación entre dichos objetos, denominado *paso de mensajes*.

Según la metodología POO en las aplicaciones no existirá ningún algoritmo global que dirija la ejecución del programa o del sistema–software, sino una cooperación entre varios objetos que intercambian información y peticiones a través de mensajes. Se puede pensar en la POO como una analogía de una comunidad humana que resuelve sus problemas de forma cooperativa.

Otros conceptos útiles de la POO son la *abstracción* (concepto de *clase*), y la *herencia* (concepto de *subclase*). Estos conceptos permiten especificar, clasificar y reutilizar las descripciones de los objetos de una forma comprensible y autodocumentada para el desa rrollador de software.

Puedes encontar más detalles sobre los conceptos anteriores, aplicados al lenguaje Java, en la documentación del "Seminario sobre Programación Multihebra en Java" (Sem_java.pdf).

2.2.2 El lenguaje de programación Java

Java comenzó como el resultado de la búsqueda de un lenguaje para programar dispositivos empotrados (*embedded control devices*) [Lea, 2001], [Hortsmann and Cornell, 2012]. Se buscaba un lenguaje con un conjunto "mínimo" de instrucciones a nivel máquina, muy eficiente, que generase binarios muy *ligeros*[1] y *transportables*[2]. Llamaron a este lenguaje *Oak* ("roble" en inglés), para dar a entender la robustez y confiabilidad que se podía esperar de las aplicaciones desarrolladas con él.

Los informáticos de Sun se dieron cuenta que un derivado del lenguaje Oak, neutral respecto de arquitecturas, era ideal para programar en Internet. En consecuencia, definieron un nuevo lenguaje al que llamaron Java y llamaron applets al código Java *recortado* que podía autoejecutarse cuando un navegador lo encuentra empotrado en una página Web.

> Definición de Java:
> A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded and dynamic language.

[1] las librerías para enlazar no hicieran aumentar demasiado el tamaño del programa ejecutable
[2] ejecutables en distintos empotrados sin necesidad de recompilación

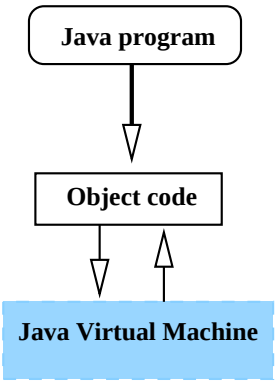


Figura 2.1: Plataforma de ejecución de código con Java

Puedes encontrar más información sobre Java en las siguientes referencias [Sánchez-Allende, 2005], [Eckel, 2009], [Darwin, 2004]

La Máquina Virtual Java (JVM)

Se trata de una *máquina virtual de proceso nativo*, es decir, es capaz de interpretar y ejecutar instrucciones en un código especial (el bytecode Java), que es generado por el compilador del lenguaje: orden javac.

Cada plataforma hardware específica implementará la Máquina Virtual Java (en inglés Java Virtual Machine, JVM), de tal forma que el código Java puede ser transportado y ejecutarse sin problemas en diferentes plataformas [Hortsmann and Cornell, 2008]. De ahí el famoso axioma que crean los introductores de Java para describir la principal utilidad de su producto: "escríbelo una vez, ejecútalo en cualquier parte" –"Write once, run anywhere".

La JVM es en sí misma un objeto, aunque muy especial ya que puede "fingir" ser cualquier otro objeto –o más formalmente podemos decir que se trata de una máquina universal de Turing.

Por otra parte, en una JVM secuencial sería imposible simular directamente objetos concurrentes que interactúan entre sí. Tampoco sería sencillo simular todo el paso de mensajes que da lugar la ejecución de una aplicación que utilice dichos objetos concurrentes. Por tanto, desarollar un software que sólo realice procesamiento secuencial limita mucho los aspectos de diseño de alto nivel que se pueden llegar a expresar con la plataforma de desarrollo Java.

Para resolver los problemas de inadecuación entre los objetos y la secuencialidad se pueden proponer varias soluciones:

- **Objetos activos**: cada objeto de la aplicación es autónomo y puede ser simulado mediante una JVM secuencial independiente. De esta manera, los distintos objetos pueden residir en distintas máquinas e incluso comunicarse a través de una red.
- **Modelos activos**: se obtienen simplificando los objetos activos para que puedan ser expresados más fácilmente utilizando hebras. En este modelo conviven las hebras y los

objetos. Una JVM puede estar compuesta de múltiples hebras, cada una de las cuales actúa de una manera similar a una JVM secuencial.

El lenguaje de programación Java opta por la segunda alternativa (*modelo mixto* formado por objetos pasivos y hebras). La razón de esta elección se debe a que las hebras pueden ser configuradas de forma eficiente tanto en máquinas con 1 solo procesador como en multiprocesadores, sin dependencias importantes de la arquitectura si asumimos un modelo *lógico de paralelismo* o *concurrencia*. De esta manera, los programas concurrentes en Java dejan los detalles de configuración –asignación de hebras a procesadores, por ejemplo– a la JVM de cada máquina y al sistema operativo subyacente. Compilando los programas a un código intermedio (bytecodes) que ejecuta la JVM se consigue, por tanto, incrementar la portabilidad de las aplicaciones concurrentes –a expensas de perder algo de eficiencia– respecto de una compilación que generase código nativo para cada plataforma de ejecución.

2.3 Tutorial de hebras de Java

En esta primera parte de la práctica se pretende adquirir los conocimientos para crear hebras en programas Java de acuerdo con las distintas alternativas que nos ofrece este lenguaje:

- Creándose una subclase de la clase Thread de Java.
- Implementando la interfaz Runnable en una clase cuyos objetos convertimos después en hebras.
- Perfeccionando el caso anterior mediante la programación de un método constructor que crea el objeto y lo convierte en una hebra directamente.

2.3.1 Extendiendo a la clase Thread

Creamos una clase que *extienda* a la clase `Thread` y *redefina* el método público `run()`. El método `run()` es el código que la hebra va a ejecutar cuando sea lanzada. El constructor de la clase (método `A1()`) redefine y llama al constructor de la superclase (`Thread`), permitiendo así asignar un nombre a la hebra (cuando ésta sea creada), tal como puede verse en la siguiente plantilla:

```
class A1 extends Thread {
  long siesta=0;
  public A1 (String nombre, long siesta) {
    super (nombre);             // llama al constructor de la superclase
    this.siesta = siesta;       // asigna un valor para siesta
  }
  public void run () {
    while (true) {
      System.out.println ("Hola, soy " + this.getName());
      if (siesta>0)
        try {
          Thread.sleep (siesta);  // duerme la siesta
        } catch (InterruptedException e) {
          System.err.println ("me fastidiaron la siesta!");
        }
    }
  }
}
```

En otro lugar creamos una instancia (`a`) de la clase (`A1`) definida anteriormente. De esta forma se crea un objeto hebra, es decir, una instancia de una subclase de `Thread`. Llamamos al método `start()` para la instancia creada. Esta llamada hace que el método `run()` se ejecute como una nueva hebra.

```
class ej1 {
  public static void main (String[] args) {  // metodo principal
    long siesta=0;
    try {
      siesta = Long.parseLong(args[0]);
    } catch (Exception e) {
      System.out.println ("Error: debe suministrar un valor de siesta");
      System.exit (1);
    }
    A1 a = new A1 ("la hebra a", siesta);  // crea una instancia a de A1
    a.start ();                            // llama al metodo start() de a
  }
}
```

2.3.2 Implementando el interfaz `Runnable`.

El problema de crear hebras de Java en la forma anterior es que no permite que la nueva clase herede de otras clases diferentes de `Thread`, ya que Java no permite herencia múltiple. La solución es hacer que la nueva clase implemente la interfaz `Runnable` en lugar de extender directamente a la clase `Thread`.

Creamos una clase que implemente el interfaz Runnable. En el constructor de la nueva clase llamamos al constructor de la superclase `OtraClase`: ahora la llamada a `super()` no es una llamada al constructor de la clase `Thread`, puesto que no estamos heredando de ella (por esta razón no tiene sentido dar el nombre como argumento a la llamada `super()`).

```
class A2 extends OtraClase implements Runnable {
  long siesta=0;
  public A2 (long siesta) {
    super ();
    this.siesta = siesta;
  }
  public void run () {
    while (true) {
      System.out.println ("Hola, soy " + Thread.currentThread().getName());
      if (siesta>0)
        try {
          Thread.sleep (siesta);
        } catch (InterruptedException e) {
          System.err.println ("me fastidiaron la siesta!");
        }
    }
  }
}
```

Primero creamos un objeto como una instancia de la clase `A2` y después una hebra, es decir, una instancia de la clase `Thread`. En el constructor indicamos como argumentos la referencia (`a`) a la instancia de la clase `A2`, creada en el paso anterior, y el nombre (`t`)de la nueva hebra creada. La referencia a la instancia de `A2` indica el objeto cuyo método `run()` va a ser ejecutado por la nueva hebra creada. Por último, llamamos al método `start()` de la hebra creada, que comenzará a ejecutarse de forma asíncrona con el programa.

```
class ej2 {
  public static void main (String[] args) {
    long siesta=0;
    try {
      siesta = Long.parseLong(args[0]);
    } catch (Exception e) {
      System.out.println ("Error: debe suministrar un valor de siesta");
      System.exit (1);
    }
getPriority(     A2 a = new A2 (siesta);   // crea una instancia a de A2
    Thread t= new Thread (a,"la hebra a"); // crea una hebra t ejecutando a
    t.start ();                            // llama al metodo start() de t
  }
}
```

2.3.3 Prioridades y planificación

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_PRIORIY` .

La prioridad actual de una hebra puede obtenerse invocando el método `getPriority()`, y se modifica con `setPriority()`. En el ejemplo siguiente creamos una hebra y decrementamos su prioridad antes de llamar a `start()`:

```
hebra = new Thread (objeto);
hebra.setPriority (hebra.getPriority() - 1);
hebra.start();
```

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. El quantum de tiempo suele estar en el orden de los 100 milisegundos. Las versiones de JDK actuales implementan la planificación por tiempo compartido.

Pero las versiones JDK para Linux realmente no realizan tiempo compartido. Esto es debido a que una hebra se ejecuta hasta que: termina, para, se bloquea, suspende, ejecuta `yield()` o aparece otra hebra ejecutable de mayor prioridad.

Para evitar el problema de planificación de hebras en Linux podemos implementar un pseudo-planificador de tiempo compartido mediante una hebra de máxima prioridad que se bloquea durante un quantum de tiempo mediante llamadas a `sleep`. Cuando la llamada a `sleep()` vuelve, la hebra pasa a estar ejecutable, y al tener máxima prioridad, desplaza a la que en ese momento se está ejecutando. Cuando la hebra planificadora vuelva a bloquearse, pasará a estar activa otra hebra del programa. Ocasionándose de este modo un desplazamiento del procesador de la hebra activa cada cierto periodo de tiempo (o "quantum").

```
public class Scheduler implements Runnable {
   private int timeSlice = 0; // milliseconds
   private Thread t = null;
   public Scheduler (int timeSlice) {
      this.timeSlice = timeSlice;
      t = new Thread(this, "scheduler");   // crea hebra
      t.setPriority(Thread.MAX_PRIORITY);  // indica maxima prioridad
      t.setDaemon(true);                   // creamos la hebra como demonio
      t.start();                           // lanza hebra
```

```
    }
    public void run() {
       while (true)
          try {
            Thread.sleep (timeSlice);
           } catch (InterruptedException e) {
            System.out.println (t.getName()+ " interrupted time slicing!!!");
           }
    }
  }
```

2.3.4 Ejercicio 1

El ejercicio que se propone se ha de crear un array de hebras de acuerdo con los esquemas de creación de hebras vistos en los subapartados anteriores. Los comentarios entre un par de asteriscos (**) indican el código que el alumno debe completar.

```
class B1 extends Thread {
  int vueltas=0;
  long siesta=0;
  boolean yielD=false;
  public B1 (String nombre, int vueltas, long siesta, boolean yielD) {
    super (nombre);           // llama al constructor de Thread
    this.siesta = siesta;
    this.vueltas = vueltas;
    this.yielD = yielD;
  }
  public void run () {
    for (int i=0; i<vueltas || vueltas==0; i++) {
      System.out.println ("Hola no."+i+", soy " +
              // ** llamar a getName para obtener nombre ** );
      if (siesta > 0)
        try {
          // ** llamar a sleep con valor siesta **
        } catch (InterruptedException e) {
          System.err.println (Thread.currentThread().getName()+
                              ": me fastidiaron la siesta!");
        }
      else if (yielD)
        // ** llamar a yield para pasar la hebra a estado ejecutable **
    }
  }
}
class p21 {
  public static void main (String[] args) {
```

```
GetOpt opt = new GetOpt (args, "Un:p:s:v:t:y");
Scheduler scheduler = null;            // referencia al planificador
opt.optErr = true;
int opcion = -1;
int nHebras = 1;                           // numero de hebras a crear
int prio = Thread.currentThread().getPriority();  // prioridad hebras
int sched = 0;                 // quantum de tiempo para planificador
long siesta = 0;               // valor de tiempo para sleep
int vueltas = 0;               // numero de iteraciones de las hebras
boolean yielD = false;         // indica si llamar o no a yield

while ((opcion = opt.getopt()) != opt.optEOF)  // procesa argumentos
  if ((char)opcion == 'U') {
    System.out.println("Uso:  -n <numero de Hebras> ");
    System.out.println("      -p + | -      // prioridad relativa a main");
    System.out.println("      -s <tiempo de planificacion>");
    System.out.println("      -v <numero de vueltas>");
    System.out.println("      -t <tiempo de siesta> | 0");
System.out.println("      -y            // hacer yield() al iterar?");
    System.exit (1);
  }
  else if ((char)opcion == 'n')
    nHebras = opt.processArg (opt.optArgGet(), nHebras);
  else if ((char)opcion == 'p') {
    char signo = opt.optArgGet().charAt(0);
    if (signo == '+')
      prio = Thread.currentThread().getPriority() + 1; // sube prioridad
    else if (signo == '-')
      prio = Thread.currentThread().getPriority() - 1; // baja prioridad
    else
      prio = Thread.currentThread().getPriority();    // misma prioridad
  }
  else if ((char)opcion == 's') {
    if ((sched = opt.processArg (opt.optArgGet(), sched)) > 0)
      scheduler = new Scheduler (sched, true);  // crea planificador
  }
  else if ((char)opcion == 'v')
    vueltas = opt.processArg (opt.optArgGet(), vueltas);
  else if ((char)opcion == 't')
    siesta = opt.processArg (opt.optArgGet(), siesta);
  else if ((char)opcion == 'y')
    yielD = true;
  else System.exit(1);

System.out.println ("nHebras="+nHebras+", prio="+prio+", sched="+sched
          +", vueltas="+vueltas+", tsiesta="+siesta+", yield="+yielD);
```

```
    // ** declarar un array la clase B1 **
    for (int i=0; i<nHebras; i++) {
      b[i]= new B1 ("la hebra b-"+i, vueltas, siesta, yielD);
      // ** crear una nueva hebra b[i] de la clase B1 **
      // ** indicar prioridad para b[i] **
      // ** lanzar la hebra (llamar a start) **
    }
    try {
      Thread.sleep (1000);
      System.out.println ("main(): termine de dormir");
    } catch (InterruptedException e) {
      System.out.println ("main(): me fastidiaron la siesta!");
    }
    if (vueltas == 0)  // si las hebras se ejecutan en un bucle infinito
      for (int i=0; i<nHebras; i++)
        // ** invocar stop para la hebra b[i] **
    else               // si las hebras ejecutan un numero de vueltas finito
      for (int i=0; i<nHebras; i++)
        try {
          // ** invocar join para la hebra b[i] **
    } catch (InterruptedException e) {
          System.out.println ("no puedo hacer join con hebra b-"+i);
        }
  }
}
```

2.3.5 Configuración de librerías

Para poder ejecutar el compilado: `p21.class`, el programa `java` tiene que encontrar los ficheros `GetOpt.class` y `Scheduler.class`. La ruta para encontrarlos ha de estar incluída dentro de la variable de entorno `CLASSPATH`. En dicha variable de entorno tienen que aparecer, al menos, 2 directorios:

- El directorio donde se encuentran las clases del lenguaje `Java`.
- El directorio de trabajo (.) para que pueda ejecutar las clases propias.

```
setenv CLASSPATH /<directorio que contiene el j2sdkX.Y.Z/lib/:.
```

Para poder realizar esta práctica necesitarás las clases `Scheduler` y `GetOpt`. Para poder usarlas sin problema haremos lo siguiente:

- Crea en tu directorio de trabajo un subdirectorio llamado `Utilities` (con U mayúscula), que contenga los ficheros: `Scheduler.java` y `GetOpt.java` que vienen con la plantilla.
- Entra en el subdirectorio `Utilities` y compila (`javac`): `Scheduler.java` y `GetOpt.java`.

2.3.6 Lista de actividades

Se han de realizar las siguientes actividades en el orden indicado:

1. Completar la plantilla proporcionada, compilar y probar el programa.
2. Crear un subdirectorio `Utilities` en el directorio de trabajo para guardar los ficheros: `Scheduler.java` y `GetOpt.java`
3. Incluir la claúsula `import Utilities.*;` en el programa. Dicha claúsula importa los contenidos del *package Utilities*, es decir, los ficheros contenidos en el subdirectorio del mismo nombre.
4. Comprobar que el programa es correcto, verificando que todas las hebras terminan, después de realizar el número de iteraciones que indicado al ejecutarlo (opción `-v <numero de vueltas>`).

2.4 Exclusión mutua en Java

Para poder tener aplicaciones "seguras" en Java cada objeto se protege contra posibles violaciones de integridad de sus datos por parte de las hebras del programa. Las técnicas de exclusión que vamos a introducir preservan la certeza de los *invariantes globales* del objeto, antes y después de la ejecución de sus métodos. De esta manera se evitan los efectos desastrosos que resultarían si las hebras del programa pudieran modificar los valores de las variables propias de un objeto cuando este posee un estado momentáneamente incoherente. Piénsese, por ejemplo, que una segunda hebra consumidora se le permitiera retirar un dato en el buffer antes de que la primera hubiera terminado de ejecutar el método y no se hubiera modificado todavía el valor del puntero `frente`. El resultado podría ser que el mismo dato se obtendría 2 veces y el puntero `frente` retrocedería 2 posiciones, perdiéndose un dato almacenado en el buffer. Para evitar esto hay que asegurar, con la programación de los métodos del objeto, que el invariante global del objeto se vuelve a cumplir [3] antes de permitir la ejecución de ningún método por las hebras.

Las técnicas de programación que propone Java para conseguir la exclusión mutua evitando que las múltiples hebras modifiquen concurrentemente representaciones de objetos:

1. Eliminar la necesidad de control de la exclusión de formal total o parcial asegurándose que los métodos nunca modifiquen la representación de un objeto.
2. Asegurar dinámicamente que sólo 1 hebra pueda acceder a la vez al estado del objeto, protegiendo los objetos con cerrojos.
3. Asegurar estructuralmente que sólo 1 hebra pueda utilizar un objeto dado mediante la ocultación o la restricción del acceso al mismo.

   En esta práctica vamos a revisar la segunda posibilidad: la sincronización protegiendo el estado del objeto, asegurando el acceso de las hebras de 1 en 1.

[3] en el caso de nuestro ejemplo:`frente= (atras + elementos -1)mod N +1)`

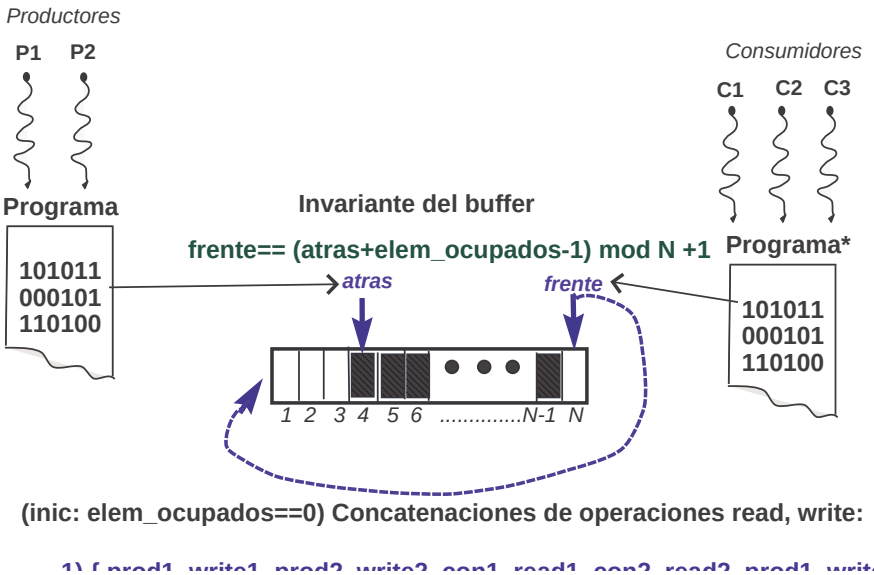


Figura 2.2: Representación del funcionamiento del buffer en un *productor-consumidor*

2.4.1 Sincronización

Se va a presentar un resumen de las distintas formas de conseguir la sincronización en el acceso a lo objetos por parte de la hebras en Java. Así como algunas observaciones con respecto al uso correcto de la palabra clave `synchronized` del lenguaje Java.

Objetos y cerrojos

Los objetos de la clase `Object` y sus subclases ya poseen 1 cerrojo –no es necesario declararlo– sólo hay que saber cómo activarlo para que proteja a bloques de código o a métodos completos de los objetos frente al acceso concurrente de las hebras.

El cerrojo de un objeto solamente puede ser adquirido por 1 sola hebra en cada momento. Todas las instancias de clases de Java son `Objects`. Los tipos escalares `int, float,...` no son `Objects` y sólo se pueden *bloquear* mediante su inclusión en objetos de Java.

No hay estructuras que bloqueen simultáneamente objetos múltiples en una sola operación atómica. El bloqueo de un array de `Objects` no bloquea automáticamente a todos sus elementos. Por tanto, un array de objetos que almacene elementos escalares posee cerrojos, pero sus elementos escalares individuales no.

Métodos y bloques sincronizados

El cualificador `synchronized` sirve para hacer que un bloque de código o un método sea protegido por el cerrojo del objeto. La sincronización de bloques toma como argumento el objeto

que bloquea. Esto permite que cualquier método bloquee a cualquier objeto que pertenezca a su ámbito –es decir, que sea visible según las claúsulas de importación de la clase a la que pertenece o pertenezca a su mismo paquete y/o mismo fichero. Para bloquear el objeto actual se utiliza, como argumento de un bloque `synchronized`, la referencia `this`.

La sincronización de bloques:

```
void synchronized (this) // * cuerpo * //
```

es equivalente a la sincronización de métodos:

```
synchronized void f() // * cuerpo * //
```

El modificador `synchronized` no se hereda automáticamente cuando las subclases redefinen métodos de la superclase y los métodos de las interfaces no se pueden declarar como `synchronized`. Los constructores tampoco se pueden declarar como `synchronized`, aunque se pueden programar bloques sincronizados dentro de estos.

Los métodos de instancia sincronizados de las subclases utilizan el mismo cerrojo que sus *superclases*, es decir, cuando una hebra llama a los métodos sincronizados de un objeto adquiere el cerrojo del objeto, no importa que dicho método sea propio de la clase a la que pertenece el citado objeto o a su superclase.

Adquisición y liberación de cerrojos

Para ejecutar un bloque o un método sincronizado, cada hebra debe adquirir primero el cerrojo del objeto, debiendo esperar si el cerrojo ha sido ya adquirido por otra hebra. Una hebra avanza su ejecución por un método `synchronized` si el cerrojo está libre o la hebra ya posee el cerrojo de ese objeto[4], en otro caso se bloquea.

Los métodos que no se programan con el atributo `synchronized` pueden ejecutarse en cualquier momento, incluso si un método `synchronized` del mismo objeto se estuviera ejecutando en ese momento. Es decir, `synchronized` no es equivalente a *atómico*, sino que la sincronización se puede utilizar para conseguir la atomicidad.

Cuando una hebra desbloquea un cerrojo, cualquier otra hebra que estuviese bloqueada esperando el cerrojo puede adquirirlo, incluso la misma hebra que lo deja libre si da con otro método `synchronized`. No hay garantías de *equitatividad* en el desbloqueo de las hebras que esperan adquirirlo cuando un cerrojo se queda libre.

2.4.2 Miembros estáticos

El bloqueo asociado a los objetos de Java no protege automáticamente el acceso a los campos `static` de la clase a la que pertenecen esos objetos, ni tampoco a los de cualquiera de sus

[4] A diferencia de lo que ocurre con las hebras POSIX, un método `synchronized` puede hacer una llamada a otro método `synchronized` del mismo objeto sin que se paralice su ejecución.

superclases. El acceso a los campos `static` de una clase se ha de proteger utilizando métodos y bloques `synchronized static` .

Un cerrojo estático asociado con una clase no está relacionado con el cerrojo estático de otra clase ni con el de su *superclase*. Por tanto, programar un método `synchronized static` en una subclase que intente proteger los campos `static` declarados en una superclase es totalmente inseguro.

2.4.3 Cómo programar objetos concurrentes en Java

La estrategia más segura consiste en programar siempre objetos *completamente sincronizados*, es decir, aquellos objetos que cumplan las siguientes condiciones:

- Todos sus métodos son declarados como `synchronized`.
- No hay campos públicos u otras violaciones de la encapsulación de los objetos de esa clase.
- Todos los métodos son finitos[5], es decir, no contienen bucles infinitos o recursión ilimitada, por lo que es seguro que los cerrojos se liberan alguna vez.
- Todos los campos se inicializan a un estado coherente con los invariantes del objeto en los métodos constructores.
- El estado del objeto cumple con los invariantes al principio y al final de la ejecución de cada método, incluso en la presencia de excepciones.

2.4.4 Ejercicio 2: array ampliable

Sin hacer uso de la sincronización, una instancia de esta clase no se podría utilizar con fiabilidad en contextos concurrentes conformados por muchas hebras que llaman impredeciblemente a los métodos de la citada clase. Por ejemplo, se podría plantear un conflicto de lectura/escritura si se admite la ejecución de un método con un acceso en medio de una operación para borrar este último elemento. Y se podría producir un conflicto de escritura/escritura si se realizan concurrentemente –sin ninguna sincronización– 2 operaciones de añadir elementos al vector, en cuyo caso el estado del array de datos sería muy difícil de predecir.

En el ejercicio que se propone se ha de crear una instancia del array ampliable y probar su funcionamiento en un programa concurrente con hebras que accedan y añadan elementos al citado array. Los comentarios entre un par de asteriscos (**) indican el código que el alumno debe completar.

[5]lo que se programa en *bucle infinito* son las hebras que llaman a esos métodos

```
class ArrayAmpliable{
  protected Object[] datos;
  protected int tamanio= e;
  //INV: 0<=tamanio<= datos.length
  public ArrayAmpliable(int capacidad){
    datos= new Object[capacidad];
  }
  public ** int tamanio(){
    return tamanio;
  }
  public ** Object leer(int i){
    **

       return ** //ha de devolver el elemento del array
                 //correspondiente al indice "i"
    }
  public synchronized void aniadir(Object x){
    **

     datos[**]= x;
  }
  public synchronized void borrarUltimo(){
   **
  }
}
```

2.4.5 Lista de actividades

Se deben realizar las siguientes actividades en el orden indicado:

1. Completar la plantilla proporcionada, compilar y probar el programa.
2. Comprobar que el programa es correcto: verificar que todas las hebras de cada programa terminan, después de realizar el número de iteraciones indicado al ejecutarlo (opción `-v <numero de vueltas>`).

2.4.6 Documentación a entregar

Los alumnos redactarán un documento donde se incluya la siguiente información:

1. Código fuente completo de las plantillas que se hayan rellenado.
2. Salidas de los programas (para un número significativo de *hebras* y *vueltas*).

2.5 "Monitores" de Java

Para construir en Java una construcción sintáctica parecida a un monitor hemos de crear una clase con sus métodos sincronizados. De esta forma solamente una hebra podrá ejecutar en cada momento los métodos del objeto.

El siguiente ejemplo muestra un monitor que implementa un contador:

```
class Contador {                  // monitor contador
  private int actual;
  public Contador (int inicial) {
    actual = inicial;
  }
  public synchronized void inc () {
    actual++;
  }
  public synchronized void dec () {
    actual--;
  }
  public synchronized int valor () {
    return actual;
  }
}

class Usuario extends Thread {   // clase hebra usuaria
  private Contador cnt;
  public Usuario (String nombre, Contador cnt) {
    super (nombre);
    this.cnt = cnt;
  }
  public void run () {
    for (int i=0; i<1000; i++) {
      cnt.inc ();
      System.out.println ("Hola, soy " + this.getName() +
                          ", mi contador vale " + cnt.valor());
    }
  }
}

class EjemploContador {          // principal
  final static int nHebras = 20;
  public static void main (String[] args) {  // metodo principal
    Contador cont1 = new Contador (10);
    Usuario hebra[] = new Usuario[nHebras];
    for (int i=0; i<nHebras; i++) {
```

```
        hebra[i] = new Usuario ("la hebra-"+i, cont1); // crea hebras
        hebra[i].start();                              // lanza hebras
      }
    }
  }
```

2.5.1 Operaciones de notificación y espera

Hasta la denominadaJava 2 Platform Standard Edition 5.0 (ó J2SE 5.0), que incluye una biblioteca denominada Concurrency Utilities, sólo se permite una única cola de condición implícita en el programa donde se bloquean las hebras que ejecutan la operación `wait()`. En lo que sigue nos centraremos en la programación de los objetos con todos sus métodos sincronizados y utilizando las operaciones `notify(), notififyAll(), wait()` anteriores a la API de J2SE 5.0. Posteriormente, realizaremos una introducción fundamental a las facilidades para programación concurrente programando con `Concurrency Utilities`.

Los métodos `wait()`, `notify()` y `notifyAll()` implementan los mecanismos de espera y notificación de los monitores Java. 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.

Operación `wait()`

La llamada a la operación `wait()` provoca que la hebra actual se bloquee y sea colocada en una cola de espera asociada al objeto *monitor*. El cerrojo del objeto se abre para que otras hebras puedan ejecutar métodos del *monitor*. Sin embargo, otros cerrojos poseídos por la hebra suspendida son retenidos por ésta.

Operación `notify()`

La llamada a `notify()` provoca que, si hay más de una hebra bloqueada en la cola de espera única, se escoja una cualquiera de forma arbitraria, y se saque de la cola, pasando ésta al estado preparado.

La hebra que invocó `notify()` seguirá ejecutándose dentro del monitor. La hebra notificada deberá adquirir el cerrojo del objeto para poder ejecutarse. Esto significará esperar al menos hasta que la hebra que invocó `notify()` abra el cerrojo, bien por que haga una llamada a `wait()`, o bien porque termine de ejecutar el método.

La hebra notificada no tiene prioridad alguna para ejecutarse en el monitor. Puede ocurrir que, antes de que la hebra señalada pueda volver a ejecutarse, otra hebra adquiera el cerrojo del monitor. La invocación al método `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.

Ejemplo: buffer circular seguro

El siguiente código muestra la programación en Java de un monitor que implementa un *buffer limitado*, que puede ser utilizado para resolver el problema de los productores y consumidores.

```
class Buffer {
private int numElementos = 0;
private double[] buffer = null;
private int atras= 0;
private int frente= 0;
private int contExt= 0;
private int contDep= 0;
private final int MAX;

public Buffer(int numElem){
//Para solo 1 productor y solo 1 consumidor
//El argumento numElem indica el
//tamanio maximo del buffer
this.numElementos=0;
this.MAX= numElem;
//para almacenar
this.buffer= new double[MAX];
contExt=0;
contDep=0;
frente= 0;
atras= 0;
}
//otros metodos, despositar y extraer
public synchronized double extraer(int id){
 double salida;
 int i;
//hay un consumidor esperando mas
 contExt++;
 while(numElementos==0){
   try{
     System.out.println("El consumidor "+
 //1:              id+" esta esperando");
     wait();
   }
   catch(InterruptedException ex){
      System.err.println("Se ha producido
                un error al extraer: "+ex);
   }
 }
 contExt--;
 salida= buffer[frente];
 numElementos--;
 frente= (frente +1)%numElementos;
 if (contDep >0)
    notifyAll();
 return salida;
}

public synchronized void depositar(
                                  double valor,
                                  int id){
  double salida;
  int i;
  //hay un productor esperando mas
  contDep++;
  while(numElementos>= MAX){
    try{
    // 2:
      System.out.println("El productor "
                         +id+
                         " esta esperando");
      wait();
    }
    catch(InterruptedException ex){
     System.err.println("Se ha producido
                         un error al
                         depositar: "+ex);
    }
  }
  contDep--;
  bufer[atras]= valor;
  numElementos++;
  frente= (frente +1)%numElementos;
  if (contExt >0)
     notifyAll();
  return null;
 }
}//fin de la clase Buffer
```

Se cumple que una hebra consumidora se bloqueará si el buffer está vacío (ver (1:)). Cuando finalmente se ejecute y extraiga 1 elemento del buffer, el buffer no puede estar lleno y, por consiguiente, hay que notificar a la hebra productora que nada le impide avanzar. Se llama a la operación `notifyAll()` para evitar bloqueos no deseados, desbloqueando a todas las hebras de

la cola única. Puesto que la hebra productora y consumidora se bloquean en la misma cola, la llamada a `notify()` despertará 1 hebra suspendida en la cola y pudiera no ser la productora.

De forma análoga al caso anterior, la hebra productora se bloqueará si el buffer está lleno (ver (2:)). Por consiguiente, se ha de llamar a la operación `notifyAll()`, ya que el número de elementos en ese momento pasa a valer `numElementos-1`.

2.5.2 API Java 5.0 para programación concurrente

Los monitores de las distribuciones de Java anteriores a la JDK[6] 5.0 (*Tiger*), que utilizan notificaciones para sincronizar internamente la ejecución concurrente de sus métodos, suponen un estilo de programación con monitores de muy bajo nivel y muy restrictiva. Dichos monitores y las primitivas asociadas (`notify(), wait()`, etc.) son adecuados para desarrollar programas concurrentes muy básicos. Necesitamos unos *bloques de construcción* de más alto nivel de abstracción para poder programar sistemas concurrentes avanzados. Esto es cierto sobre todo cuando se pretenden desarrollar aplicaciones masivamente concurrentes, que exploten las posibilidades de los sistemas multiprocesador y multicore actuales de la forma más completa que sea posible.

La distribución JDK 5.0 para la plataforma Java 2 incorporó nuevas primitivas concurrentes y clarificó bastante el anterior modelo de concurrencia de Java –el modelo que hemos estudiado en esta sección 2.5.

La mayoría de las primitivas mencionadas han sido incluídas en los nuevos paquetes *java.util.concurrent*. También se encuentran nuevas estructuras de datos concurrentes en la infraestructura *Java Collections*.

Las principales mejoras para programación currente recientemente incorporadas a las distribuciones actuales de Java:

- Un nuevo objeto–*cerrojo lock*, que soporta instrucciones de bloqueo similares a las de las hebras POSIX, útil para simplificar la programación de muchas aplicaciones concurrentes.
- Los *ejecutores* que definen una API de alto nivel para lanzar y gestionar las hebras. Ofrecen la gestión de *pools de hebras* adecuados para programación de aplicaciones concurrentes de gran escala.
- Las *colecciones concurrentes*, que hacen mucho más sencillo la gestión de grandes colecciones de datos, y pueden reducir la necesidad de sincronización entre las hebras.
- Las variables atómicas, que poseen características para minimizar la sincronización y ayudan a evitar errores de consistencia de memoria.
- `ThreadLocalRandom` (incluída en JDK 7), que proporciona una generación eficiente de números pseudo-aleatorios que pueden ser utilizados en aplicaciones multienhebradas.

[6] Java Development Kit: incluye todas las herramientas software para desarrollar en una plataforma Java

Cerrojos y variables condición

La operación `lock()` (ver figura 2.3) deja la hebra que la llama esperando de forma ininterrumpible hasta que el cerrojo se quede libre. `LockInterruptibly()` se comporta de una manera análoga a la operación `lock()`, excepto que la espera podría ser interrumpida por otra hebra o por el sistema. El método `TryLock()`, después de ser llamado por una hebra, devuelve el valor `true` si el `lock` se encuentra disponible. Este método admite como argumentos un plazo de tiempo (*timeout*) y la unidad en la que se mide dicha espera. Devuelve el valor `true` si el `lock` se convierte en disponible durante dicho *timeout*. El método `unlock()` convierte en disponible al cerrojo para que pueda ser adquirido por otra hebra.

Anteriormente a J2SE 5.0, los monitores de Java sólo permitían una única cola de condición implícita interna donde se bloquean las hebras que ejecutan sus métodos sincronizados –al ejecutar la operación `wait()` programada dentro dichos métodos. Todas las hebras bloqueadas debían ser *despertadas* (posiblemente invocando una operación `notifyAll()`). Si después de ser despertada no se cumplía la condición de desbloqueo, la hebra tenía que ser bloqueada de nuevo. De ahí que la operación `wait()` se programe dentro del cuerpo de un bucle que re-evalúa la condición antes de dejar a la hebra seguir con su ejecución.

A partir de la J2SE 5.0 existe un método `newCondition()` (ver figura 2.3) que crea nuevas *variables condición*. Ahora se pueden declarar varias de estas variables en 1 monitor, cada una de ellas para bloquear a las hebras clientes esperando que se cumpla una condición específica. La interfaz proporciona las operaciones `await()` y sus variantes para suspender la ejecución de una hebra hasta que sea notificada por otra hebra de que alguna condición de estado podría haberse hecho cierta. Dado que el acceso a esta información es compartida entre hebras diferentes, dicha información debe ser protegida utilizando algún tipo de cerrojo que ha de estar permanentemente asociado a la condición. De ahí que toda variable `Condition` ha de ser utilizada junto con un objeto asociado conforme a `Lock`.

La característica clave asociada a la operación de esperar a una condición (`await()`) consiste en que la llamada a dicha operación libera de forma ininterrumpible el cerrojo asociado y suspende a la hebra actual, de una forma totalmente análoga a la ejecución de `Object.wait()`. Por ejemplo, supongamos que tenemos un *buffer* limitado cuyo contenido es modificado llamando a los métodos `depositar()` y `extraer()`. Si se intenta llamar el método `extraer()` cuando el buffer está vacío, entonces la hebra se bloqueará hasta que el buffer contenga algún elemento; si se intenta el método `depositar()` cuando el buffer está lleno, entonces la hebra se bloqueará hasta que llegue a haber espacio libre en el buffer. Nos gustaría que las hebras que esperan ejecutar la operación `extraer()` fueran situadas en 1 cola de espera diferente de aquellas que esperan a la operación `depositar()`. De esta forma podríamos notificar de forma totalmente independiente a una hebra que espera insertar o eliminar cada vez que aparezcan huecos o elementos en el buffer, respectivamente. Esto se puede conseguir creando dos instancias distintas de `Condition` . Por ejemplo, 2 variables condición: (1) para hacer esperar las hebras productoras a que no esté lleno el buffer y (2) para que las consumidoras esperen a que no esté vacío.

La ejecución de la operación `await()` dentro de un método libera el cerrojo asociado al monitor de forma atómica y ocasiona que la hebra actual espere hasta que: (a) otra hebra llame al método `signal()` y la primera sea escogida como la hebra que va a ser *despertada*; (b)

```
public interface Lock {

 public void lock();

 public void lockInterruptibly()
   throws InterruptedException;

 public Condition newCondition();

 public boolean tryLock();


 public boolean tryLock(long time,
                        TimeUnit unit)
        throws InterruptedException;


 public void unlock();
 }
```

```
package java.util.concurrent.locks;

public interface Condition {

  public void await()throws
              InterruptedException;
  public boolean await(long time,
                       TimeUnit unit)
        throws InterruptedException;
  public long awaitNanos(long
                         nanosTimeout)
        throws InterruptedException;
  public void awaitUninterruptible();
      // As for await, but not
      // interruptible.
  public boolean awaitUntil(
          java.util.Date deadl)
  throws InterruptedException;
      // As for await() but with
      // a timeout
  public void signal();
      // Wake up one waiting thread.
  public void signalAll();
      // Wake up all waiting threads.
}
```

Figura 2.3: Interfaces de J2SE 4.0 para cerrojos y variables condición

otra hebra llame al método `signalAll()`, ocasionando que todas las hebras que esperan sean despertadas;(c) otra hebra interrumpa la espera de la primera hebra; o bien, (d) se despierte la hebra de forma espúrea o imprevista. Cuando la operación `await()` vuelva está garantizado que la hebra que la llamó volverá a poseer el cerrojo (o `lock`) asociado a dicha variable condición.

Cómo se utiliza la API java 5.0

Los creadores de Java quisieron que la nueva API fuese muy intuitiva y fácil de manejar para aquellos que estén familiarizados con las facilidades de bajo nivel para programar monitores, basadas en métodos sincronizados y operaciones `wait()` y `notify()`. Aquí se muestran algunas reglas prácticas que sirven de guía para desarrollar código utilizando la nueva API:

API anterior

```
Object objetoMonitor;

synchronized(objetoMonitor){
  //seccion critica
}
```

Nueva API

```
Lock cerrojo;
try{
  cerrojo.lock();
  //seccion critica
}
finally{
  cerrojo.unlock();
}
```

En lugar de utilizar los monitores de bajo nivel de Java, basados en declaraciones `synchronized`, ahora se *emparedan* las secciones críticas de código con llamadas a los métodos `Lock.lock()` y `Lock.unlock()` de la clase `Lock`, que viene con el paquete `java.util.concurrent.locks.*;` Ahora las hebras de la aplicación en lugar de confiar en adquirir el monitor de bajo nivel, es decir el cerrojo implícito de todas las intancias de la clase `Object`, con la nueva API intentan adquirir un objeto *cerrojo* que se instancia a partir de la clase `java.util.concurrent.locks.Lock`. `Lock` es una interface de Java, que da lugar a pocas implementaciones:

- `ReentrantLock`,
- `ReentrantReadWriteLock.ReadLock`,
- `ReentrantReadWriteLock.WriteLock`.

En la parte de declaraciones de variables y objetos privados de la clase hay que declarar ahora `private Lock cerrojo = new ReentrantLock();`. En la API Java 5.0 este objeto cerrojo explícito nos va a proporcionar la protección que necesita un módulo monitor para poder ser utilizado con seguridad por las hebras de la aplicación concurrente.

En el código, para cada llamada que se haga a `lock()` hay que acordarse de hacer la llamada correspondiente a `unlock()` . Esta es la razón de por qué la llamada a `unlock()` se hace dentro de una claúsula `finally` dentro de un bloque `try-catch`. En conclusión, el código que ir'ıa en una sección crítica en la API anterior, protegido por `synchronized`, ahora va dentro de un bloque `try-catch-finally`.

En las secciones de código que se necesitaba utilizar `wait()` y `notify()/notifyAll()` hay que utilizar ahora *variables condición*. Con la API anterior las hebras podían esperar en colas, hasta que fueran señaladas por la ejecución de las operaciones `notify()/notifyAll()`. Puesto que ya no vamos a utilizar los monitores de bajo nivel, tenemos que declarar una *variable condición* del tipo `java.util.concurrent.locks.Condition`. `Condition` es una interface en Java. Para poder utilizar una variable condición en un programa hay que crearla a partir del objeto de tipo `Lock`, que hemos tenido que adquirir previamente a entrar en la sección crítica. Se pueden generar tantas variables condición como se necesiten. Una vez que tenemos la variable condición disponible, se puede llamar a la operación `await()` sobre ella y obtener la misma funcionalidad que teñamos con la llamada a `Object.wait()` en los monitores de bajo nivel. Normalmente, evaluaríamos una condición en un bucle `while` y dependiendo del resultado de esta evaluación, la hebra se bloquearía o no.

De una manera análoga, con `Condition.await()` hemos de evaluar dicha condición en un bucle `while` y si fuera necesario hacer esperar a la hebra, para lo cual se llamaría a la operación `Condition.await()`.

API anterior

```
//Dentro de la seccion//
//critica:
////////////////////////////////
boolean alguna_condicion;
//evaluar las razones para esperar

while(alguna_condicion){
  wait();
  //re-evaluar alguna_condicion
}
```

Nueva API

```
//Fuera de la seccion://
//critica://
////////////////////////////////
Condition variableCond =
          cerrojo.newCondition();
////////////////////////////////

//Dentro de la seccion //
//critica ://
////////////////////////////////
boolean alguna_condicion;
//evaluar las razones para esperar

while(alguna_condicion){
  variableCond.await();
  //re-evaluar alguna_condicion
}
```

El equivalente a la llamada `notify()/notifyAll()` para desbloquear a una hebra suspendida en un `wait()` se consigue ahora utilizando las operaciones `signal()/signalAll()`.

API anterior

```
//Dentro de la seccion//
//critica://
////////////////////////////////////
boolean alguna_condicion;
//evaluar las razones para
//dejar deesperar
if(alguna_condicion) {
  notify();
}
```

Nueva API

```
//Dentro de la seccion//
//critica://
////////////////////////////////////
boolean alguna_condicion;
//evaluar las razones para
//dejar de esperar
if(alguna_condicion) {
  conditionVariable.signal();
}
```

La nueva API de Java para concurrencia no modifica de ninguna manera la independencia del comportamiento de las hebras durante su ejecución. No se puede realmente desplazar a una hebra desde nuestro programa, sólo podemos señalarlas indicando qué cosas se even interrumpidas en cada momento de la ejecución y si se desatiende una *InterruptedException*, que pueda ser capturada por la hebra que está esperando.

2.6 Ejercicio 3: El problema de los fumadores

Suponiendo las mismas condiciones del problema que en la práctica 1. Para resolver correctamente usando las notificaciones de Java –versiones de JDE sin variables condición– hay

que programar varias clases y hebras y utilizar un monitor, que llamaremos `Control`, para sincronizar el estanquero con los tres fumadores.

Todos los objetos *fumadores* son creados a partir de una clase `Fumador` en la que el ingrediente que posee el fumador se pasa como argumento de uno de sus métodos.

- La actuación de los 3 fumadores y del estanquero se simulan con 1 hebra cada uno.
- Las hebras que representan a los fumadores se encuentran inicialmente bloqueadas.
- El estanquero, después de obtener los ingredientes de 2 fumadores elegidos al azar, pone 2 ingredientes en la mesa y desbloquea al fumador que posee el ingrediente que falta para liar un cigarro. Después, el estanquero se bloquea.
- El fumador que ha sido desbloqueado, retira de la mesa los 2 ingredientes de una vez, lía un cigarrillo y se lo fuma durante un periodo de tiempo de duración arbitraria –se debe generar aleatoriamente dentro de un rango de valores temporales prestablecido. Cuando termine de fumar, desbloquea al estanquero.
- Entonces, el estanquero volverá a obtener aleatoriamente otros 2 ingredientes, que pondrá en la mesa, repitiéndose de nuevo el ciclo hasta ejecutar el método de parada.

Requisitos que ha de satisfacer el programa:

- Se utiliza sólo 1 objeto monitor instanciado a partir de la clase `Control` para programar la sincronización entre las hebras de la simulación.
- No se permiten *bloques synchronized*, utilizar sólo métodos sincronizados.
- Para implementar la sincronización hay que utilizar la operación `wait()`.
- No utilizar nunca la función `Thread.sleep()` dentro de un método `synchronized` o mientras se llama a un método llamado por un método sincronizado.

2.6.1 Plantilla para el programa de los fumadores

```
import java.io.*;
import java.lang.Math;
import java.util.Random;
import Utilities.*;
//Monitor para sincronizar la actuaci\'on de los 3 fumadores y el estanquero
class Control{
//Declaraci\'on de las variables de instancia del monitor
//y de las variables estaticas finales que representan los estados del monitor:
//LISTO = 0 : para indicar a 1os fumadores que ya hay 2 ingredientes en la mesa
//ESPERANDO =1 : mientras el fumador est\'a esperando los ingredientes adecuados
//INICIO = 2 : estado inicial; el estanquero espera que las hebras fumadoras
//sean lanzadas.
```

```
private static final int LISTO= 0;
private static final int ESPERANDO= 1;
private static final int INICIO= 2;
private int estado= INICIO;
private int cont=0;
private char ingrediente='f';//valor inicial no valido
public synchronized void generar_Ingredientes(){
//** Generar aletoriamente un n\'umero del conjunto={1,2,3}
//**  asignarlo a la variable selector
  switch(selector){
case 1: this.ingrediente= 'c';//al fumador con cerillas se le autoriza a fumar
    break;
    case 2: this.ingrediente= 't';//al fumador con tabaco se le autoriza a fumar
        break;
    case 3: this.ingrediente= 'p';//al fumador con papel se le autoriza a fumar
        break;
    default: System.out.println("Error al generar los ingredientes");
    }
}
//se lee el ingrediente del fumador autorizado
public synchronized char leer_Ingrediente(){
    return this.ingrediente;
}
//se informa de qu\'e fumador est\'a liando
public synchronized void informar_Liar (char ingred){
  switch(ingred){
    case 'c':
  System.out.println("El fumador con cerillas lia el cigarro \n");
      break;
    case 't':
      System.out.println("El fumador con tabaco lia el cigarro \n");
      break;
    case 'p':
      System.out.println("El fumador con papel lia el cigarro \n");
      break;
      default:
        System.out.println("Clase fumador_Espera():
                    lo que se ha pasado no es un ingrediente de la simulacion");
    }
}
public synchronized void fumador_Espera(char ingred){
   //Los 3 fumadores han de ejecutar este metodo y se quedan esperando,
   //antes de que el fumador que tiene el ingrediente que falta pase
   //a liar 1 cigarro. Solo cuando se haya entrado aqui el tercer fumador
   //puede lanzarse la hebra "estanquero"

   //** Primero hay que ponerse a esperar a que el 'ingred' de este
   //** fumador == ingrediente
   //** del fumador autorizado y a que el estanquero notifique el
   //** estado== LISTO
```

```
      //** Los fumadores han de notificar a los dem\'as fumadores para que
      //** avancen si pueden.
      //Se informa ahora de qu\'e fumador pasa a liar el cigarro
      informar_ liar(ingred);
   }
   public synchronized void Terminar(){
      //El fumador avisa que ha terminado de fumar y
      System.out.println("El fumador termina de fumarse el cigarro\n");
      //** y notifica para que el estanquero ponga otros 2 ingredientes en la mesa.
    }
   public synchronized void actuacion_Estanquero(){
   char autorizado;
      //** Codigo de sincronizacion del estanquero
      //** El estanquero ha de esperar en el estado inicial hasta que
      //** las 3 hebras fumadoras pasen al estado ESPERANDO los ingredientes.

      //El estanquero pone los 2 ingredientes en la mesa.
      //Lo hace llamando al metodo generar_Ingredientes()
       generar_Ingredientes();
       autorizado= leer_Ingrediente();
   switch( autorizado){
           case 'c':
               System.out.println("El estanquero pone en la mesa tabaco y papel");
               System.out.flush();
               break;
           case 't':
               System.out.println("El estanquero pone en la mesa cerillas y papel");
               System.out.flush();
               break;
           case 'p':
               System.out.println("El estanquero pone en la mesa tabaco y cerillas");
               System.out.flush();
               break;
           default: System.out.println("Clase actuacion_Estanquero():
                                       error en el paso del parametro ingredientes");
       }
       //** Ya estan los ingredientes en la mesa y hayq ue cambiar el estado del monitor

   //** El estanquero notifica a los fumadores y se bloquea hasta que el fumador

       //** correspondiente se haya fumado su cigarro.
   }
 }

class Fumador
//** falta indicar las relaciones de herencia e implementacion de esta clase
{
 //Ahora se declaran las variables de la clase, utilizando el modificador de
 //ambito apropiado: private, public, protected...
```

```
    private volatile Thread continua;
    private char ing; //Para almacenar el ingrediente que posee el fumador
    //Declarando los metodos necesarios para crear, iniciar y parar las
    // instancias de esta clase.
    public Fumador(char ing){
    this.ing= ing;
        this.start();
    }
    public void start(){
        continua= new Thread(this);
        continua.start();
    }
    public void stop(){
        continua= null;
    }
 public void run(){
  //** Simula la actuacion del fumador en un bucle indefinido
  //** (hasta que se ejecute el metodo de parada). El tiempo que tarda
  //** en fumarse un cigarro se puede programar con:
  //**       Random randGen= new Random();
  //**       ...
  //**       int fumada =  1 + Math.abs((randGen.nextInt())%t_max_fumando)*1000;
  //**       System.out.print("...fumando..."); System.out.flush();
  //**       Thread.sleep(fumada);
  //**  Se ha de llamar a los metodos fumador_espera(ing) del monitor control
  //**  y al final al metodo Terminar() para avisar al estaquero que ya ha
  //**  terminado de fumar y pase a pener otros 2 ingredientes en la mesa.
  }
}
class Estanquero
//** falta indicar las relaciones de herencia e implementacion de esta clase
{
    //Ahora se declaran las variables de la clase. Utilizar el modificador de
    //ambito apropiado: private, public, protected...
    private volatile Thread continua;
    //Declarando los metodos necesarios para crear, iniciar y
    //parar las instancias de esta clase.
    public Estanquero(){
    this.start();
    }
    public void start(){
    continua= new Thread(this);
        continua.start();
    }
    public void stop(){
        continua=null;
    }
    public void run(){
      //** Simula la actuacion del estanquero en un bucle indefinido
      //** (hasta que se ejecute el metodo de parada).
```

```
    }
}

//********************** Clase del Principal *****************
class p22{
  static int t_max_fumando=1; //el tiempo maximo que tarda un fumador
                              //en fumar un cigarro liado
  static int t_max_simulacion= 20; //el tiempo de ejecucion de la simulacion
  //Declaracion del monitor 'Control'
    static Control m=new Control();
  public static void main(String[] args) {
    GetOpt opt= new GetOpt (args, "Ut:E:");
    Scheduler scheduler = new Scheduler(5);
    int opcion= -1
    while((opcion = opt.getopt()) != opt.optEOF) //procesa argumentos
     if ((char) opcion == 'U')
       {
        System.out.println("Uso: -t <tiempo maximo de fumada de cualquier fumador >");
        System.out.println("     -E <tiempo de ejecucion de la simulacion>");
        System.exit(1);
       }
     else if ((char) opcion=='t')
       t_max_fumando= opt.processArg(opt.optArgGet(), t_max_fumando );
     else if ((char) opcion == 'E')
       t_max_simulacion= opt.processArg(opt.optArgGet(), t_max_simulacion);
     else{ System.out.println("Los argumentos no se han obtenido correctamente");
          System.out.flush();
          System.exit(1);
     }
    //** Crear los 3 objetos fumador[i]  y estanquero
    //** (e iniciarlos, si no se ha previsto ya dentro del constructor)

   //A dormir para dejar actuar a las hebras...
    try{
      Thread.sleep(t_max_simulacion*1000);
    }
     catch(InterruptedException ex){
                 System.err.println("Interrumpido en el Principal"+ex);
             }
      //** Terminar la ejecucion de los objetos con un metodo propio de la
      //** clase Fumador y Estanquero

  //... y ahora se termina la aplicacion completa sin esperar mas.
      System.out.println();
      System.out.println("Se acabo!!!");
      System.exit(0);
  }
}
```

2.7 Ejercicio 4: Citas múltiples

En este ejercicio se pretende obtener una simulación de un esquema de sincronización que aparece con frecuencia entre grupos de procesos concurrentes asíncronos. Se denomina *cita múltiple* a una sincronización asimétrica entre 2 grupos diferentes de procesos.

Para simular la cita múltiple vamos a hacer una analogía con la actuación de un número personas de tipo A y otro de tipo B que entran en una habitación con una puerta que sólo deja pasar 1 persona cada vez. La habitación será simulada por el monitor `Habitación` y las personas por hebras que hay que programar en `Java`, de acuerdo con las siguientes condiciones:

- 1 persona de tipo A no puede salir hasta que encuentre a 10 personas de tipo B. Las personas, por tanto, sólo pueden salir de la habitación formando una *pandilla* compuesta de 1 persona A y 10 personas B.
- Recíprocamente, 1 persona B sólo puede salir de la habitación si logra integrarse en una *pandilla* compuesta de 1 persona A y 10 personas B.
- Como la puerta es estrecha, una vez que se forme una *pandilla* para salir, las personas que pertenecen a la misma abandonan la habitación de 1 en 1.
- No se permite que entre o salga ninguna otra persona mientras la *pandilla* que se ha formado esté saliendo de la habitación.

Para resolver con mayor facilidad este ejercicio se proporciona la plantilla `p23.java` Dicha plantilla se ha programado parcialmente suponiendo que se cumplen las siguientes condiciones:

- Los objetos que representan a las personas se crean a partir de 1 de las 2 clases: `PersonaA` o `PersonaB`.
- Se utiliza 1 solo objeto *monitor* instanciado a partir de la clase `Habitación` para implementar la sincronización entre las hebras.
- La clase con el método `main()` crea a todos los objetos y llama a sus constructores que, a su vez, lanzan a las hebras.
- Las variables `na1` y `nb1` representan al número de personas de tipo A y de B –respectivamente– que están dentro de la habitación en un momento dado.
- Cuando (`na1=1 y nb1`$\geq$ `10`) o (`na  0 y nb1=10`) se forma una *pandilla* que comienza a salir de la habitación. No es necesario que estas personas sean las que lleven más tiempo dentro de la habitación, sino sólo que estén dentro.
- Las variables `na2, nb2`, cuentan a las personas que están todavía dentro, pero preparadas para salir porque pertenecen a la misma *pandilla* de salida. Las *pandillas* se forman de 1 en 1, es decir, si ya hay varias personas A en la habitación, se escoge a 1 de ellas y se comienza a formar 1 sola pandilla con las personas B que vayan entrando a la habitación. Cada persona X (A o B) ejecuta el método `sale_X()` cuando sale del monitor `habitación`.

- Cuando estas variables alcanzan sus valores máximos (`na2=1` y `nb2=10`), entonces se llama al método `terminarSalida()` del monitor `Habitación`, que registra la salida efectiva de las personas de la *pandilla*:
  - Se asignan a `0` los valores de `na2` y `nb2`
  - Se disminuyen los valores de `na1` y `nb1` en `1` y `10`, respectivamente.
- La *hebra-persona* que salga en último lugar llamará al método `terminarSalida()`.

Para resolver el ejercicio no se permite usar *bloques synchronized*, sólo métodos `synchronized`. Tampoco se deben utilizar *bucles de espera ocupada* para implementar la sincronización entre las hebras del programa, sólolas notificaciones de `Java` y sus métodos `notify(), notifyAll()` y `wait()`. No utilizar llamadas a `Thread.sleep()` dentro de 1 método *sincronizado*, o mientras se ejecuta 1 método llamado por uno sincronizado.

2.7.1 Plantilla del programa de la *habitación*

```
import java.io.*;
import java.lang.Math;
import java.util.Random;
import Utilities.*;
class Habitacion{
//declaracion de las variables del monitor
    protected int na1, na2, nb1, nb2;
    public Habitacion(){
    this.na1=0;
        this.na2=0;
        this.nb1=0;
        this.nb2=0;
    //**Declara las variables adicionales que puedas necesitar
    }
    public synchronized void entra_A(){
    //** Codigo del metodo. Se sugiere el incluir en este los mensajes:
    //** System.out.println("Persona A intentando entrar...");
    //** System.out.println("...Persona A en habitacion - na="+(na1+1));
    //** para dar mejor "visualizacion" a lo que se va a realizar.

    //** Controlar que no entren As cuando esta saliendo una pandilla


        //** Controlar que solo salga 1 persona A con cada pandilla, independiente
        //** de que en el momento de salir pueda haber mas de una en la habitacion
    }
    public synchronized void entra_B()
    {
    //** Codigo del metodo. Se sugiere el incluir en este los mensajes:
    //** System.out.println("Persona B intentando entrar...");
```

```
    //** System.out.println("...Persona B en habitacion - nb="+(nb1+1));
    //** para dar mejor "visualizacion" a lo que se va a realizar.

    //** Controlar que no entren Bs cuando esta saliendo una pandilla


        //** Controlar que salgan hasta 10 Bs con cada pandilla, independiente
        //** de que en el momento de salir pueda haber mas de 10 en la habitacion
    }
    public synchronized void sale_B(){
    //** un thread-persona B sale del monitor ...

    //** Controlar que se espera a que la persona A
    //** este preparada para salir con la pandilla
    //**  y que no se nos salgan mas de 10 Bs
  }
  public synchronized void sale_A(){
   //** un thread-persona A sale del monitor ...

   //** un thread-persona B sale del monitor ...

   //** Controlar que se espera a que la persona A
   //** este preparada para salir con la pandilla
   //**  y que no se nos salgan mas de 10 Bs
  }
  public synchronized void terminarSalida(){
    //** La cuadrilla de salida abandona "efectivamente" el monitor.
    //** Aqui es donde hay que actulizar los valores de las variables
    //** na1, na2, nb1, nb2 de una sola vez.
  }

  //** otros metodos que pudieras necesitar...

}
class PersonaA //** falta indicar las relaciones de herencia e implementacion
{
//**Declarar la variables. Utilizar el modificador de
//**ambito apropiado: private, public, protected,...


//**Declarar los metodos necesarios para crear, iniciar y
//parar las instancias de esta clase.

  public void run(){

    Random randGen= new Random();//Para los numeros aleatorios
    Thread estaHebra=Thread.currentThread();
    //** Controlar que 1 hebra-persona A puede entrar varias veces a la habitacion
    //** Y que hay que pararla para que termine bien.
            try{
```

```
                //Hacemos un pequenio sleep
                //lo ponemos en terminos de seg.
                int parar = 1 + Math.abs((randGen.nextInt())%ociosoA)*1000;
                Thread.sleep(parar);
            }
            catch(InterruptedException ex){
                System.out.println("Me fastidiaron el sleep!"+ex);
            }

    }
}
class PersonaB //** falta indicar las relaciones de herencia e implementacion
{
//**Declarar las variables. Utilizar el modificador de
//**ambito apropiado: private, public, protected,...

//**Declarar los metodos necesarios para crear, iniciar y parar
//**las instancias de esta clase.
    public void run(){
        Random randGen= new Random();//Para los numeros aleatorios
        Thread estaHebra=Thread.currentThread();
        //** Controlar que 1 hebra-persona A puede entrar
        //** varias veces a la habitacion
        //** Y que hay que pararla para que termine bien.
            try{
                //Hacemos un pequenio sleep
                //lo ponemos en terminos de seg.
                int parar = 1 + Math.abs((randGen.nextInt())%ociosoB)*1000;
                Thread.sleep(parar);
            }
            catch(InterruptedException ex){
                System.out.println("Me fastidiaron el sleep!"+ex);
            }
    }
}
class p23{
    static Habitacion m= new Habitacion();
    static int A=10;
    static int B= 100;
    static int vecesA=1;
    static int vecesB=1;
    static int ociosoA=1;
    static int ociosoB=1;
  public static void main(String[] args) {
    GetOpt opt= new GetOpt (args, "UA:v:t:B:w:s:E:");
    Scheduler scheduler = new Scheduler(5);

    //** Declarando las variables y creando los objetos que necesites
    int e= 20;
    int opcion= -1;
```

```
 System.out.println("**");
 while((opcion = opt.getopt()) != opt.optEOF) //procesa argumentos
  if ((char) opcion == 'U')
    {
     System.out.println("Uso: -A <numero personas 'A' en la simulacion>");
     System.out.println("     -v <numero de veces que 1 persona 'A'
                                                    entra en la habitacion>");
     System.out.println("     -t <tiempo ocioso de persona A en habitacion>");
     System.out.println("     -B <numero personas 'B' en la simulacion>");
     System.out.println("     -w <numero de veces que 1 persona 'B'
                                                    entra en la habitacion>");
     System.out.println("     -s <tiempo ocioso de persona B en habitacion>");
     System.out.println("     -E <tiempo de ejecucion>");
     System.exit(1);
    }
  else if ((char) opcion == 'A')
    A= opt.processArg(opt.optArgGet(), A);
  else if ((char) opcion == 'v')
    vecesA= opt.processArg(opt.optArgGet(), vecesA);
  else if((char) opcion== 't')
  ociosoA= opt.processArg(opt.optArgGet(), ociosoA);
  else if ((char) opcion == 'B')
    B= opt.processArg(opt.optArgGet(), B);
  else if ((char) opcion == 'w')
    vecesB= opt.processArg(opt.optArgGet(), vecesB);
  else if((char) opcion== 's')
  ociosoB= opt.processArg(opt.optArgGet(), ociosoB);
  else if ((char) opcion== 'E')
    e= opt.processArg(opt.optArgGet(), e);
  else { System.out.println("los argumentos no se han obtenido correctamente");
         System.out.flush();
         System.exit(1);
  }
 System.out.println("comienza la simulacion con na= "+A+"  y con nb= "+B);
 //** Crear un array de hebras-personas de los 2 tipos correctamente


//** Dar tiempo a la simulacion...
 try{
   Thread.sleep(e*1000);
 }
  catch(InterruptedException ex){
               System.err.println("Siesta interrumpida en Principal"+ex);
          }

   //** Terminar la ejecucion de los objetos-hebras llamando  al metodo adecuado

   System.out.println();
   System.out.println("Se acabo!!!");
   System.exit(0);
```

```
   }
}
```

2.8 Lista de actividades para los ejercicios 3-4

El objetivo de esta segunda parte de la práctica es primero aprender a programar la sincronización entre hebras utilizando el monitor de las instancias de la clase `Object`. En segundo lugar se pretende adquirir los conocimientos fundamentales para programar con la nueva API Java 5.0, en concreto, con algunas de las primitivas incluidas en el paquete `java.util.concurrent.*`.

Se han de realizar las siguientes actividades en el orden indicado:

1. Crear un subdirectorio que llamarás `Utilities` en tu directorio de trabajo para guardar los ficheros: `Scheduler.java` y `GetOpt.java`.
2. Completa 1 de las 2 plantillas (`Control`(-fumadores) o `Habitacion`) que se han proporcionado, compila y prueba el programa.
3. Comprueba que tus programas son correctos: verifica que todas las hebras de tus programa terminan, después de realizar el número de iteraciones que indicaste al ejecutarlo (opción `-v <numero de vueltas>`).

2.8.1 Documentación a entregar

Debes elaborar un documento (PDF)que de forma clara incluya lo siguiente:

1. El código fuente completo de todas las plantillas que hayas rellenado.
2. Un listado de salida de cada programa (para un número significativo de *hebras* y *vueltas*).
3. Adicionalmente, incluye un último punto en el que comentarás razonadamente el "grado de concurrencia" que alcanza tu programa –con las limitaciones de la plantilla– y si se te ocurre algún diseño alternativo para mejorarlo, que no ha de tener necesariamente en cuenta la plantilla propuesta.

Práctica 3

Programación de algoritmos distribuidos usando un sistema de programación basado en paso de mensajes.

3.1 Resumen

El objetivo de esta práctica es adquirir los conocimientos básicos para programar con la interfaz de paso de mensajes MPI [Snir et al., 1999] y la implementación denominada Open MPI. MPI fue diseñado para ser una implementación estándar de un modelo de programación paralela basado en paso de mensajes y consiste en un conjunto de funciones que se insertan en el código fuente de los programas para realizar comunicación de datos entre procesos. Por último, se proponen varios ejercicios para programar.

3.2 Introducción

MPI (Message Passing Interface) se ha convertido en el estándar actual para la realización de aplicaciones paralelas basadas en paso de mensajes. El modelo de programación paralela y distribuida que se puede realizar con MPI es el denominado MIMD (Multiple Instruction streams, Multiple Data streams). Pero, en la práctica, MPI suele utilizar el modelo denominado SPMD (Single Program Multiple Data), un caso particular del SIMD. En el modelo SPMD todos los procesos ejecutan el mismo programa, aunque no necesariamente han de ejecutar la misma instrucción al mismo tiempo.

La programación con MPI supone un número fijo de procesos que se comunican mediante llamadas a funciones de envío y recepción de mensajes.

Diremos que MPI pretende definir un único entorno de programación, que garantice la total

transportabilidad de las aplicaciones paralelas, basado en una única interfaz e independiente de cualquier plataforma de computación [1] Por tanto, MPI proporciona al programador una colección de funciones para que éste diseñe su aplicación, sin que tenga necesariamente que conocer el hardware concreto sobre el que finalmente se va a ejecutar, ni tampoco la forma en la que se han implementado tales funciones.

Los elementos básicos de MPI son una definición de un interfaz de programación independiente de lenguajes, más una colección de implementaciones de ese interfaz (o *bindings*, como se dice sucintamente) para los lenguajes de programación más extendidos en la comunidad usuaria de computadores paralelos, es decir: Ada, C y FORTRAN.

3.3 MPI

Cualquiera que quiera utilizar MPI para desarrollar software ha de trabajar con una implementación de MPI que constará de, al menos, los siguientes elementos:

- Una biblioteca de funciones para C/C++, más el archivo de cabecera `mpi.h` con las definiciones de esas funciones y de una colección de constantes y macros.
- Una biblioteca de funciones para FORTRAN, junto con el archivo de cabecera `mpif.h` .
- Comandos para compilación, típicamente `mpicc, mpicxx, mpif77`, que son versiones de las órdenes de compilación habituales (`gcc,g++, f77`) que incorporan automáticamente las bibliotecas de funciones basadas en MPI.
  Ejemplo:

  ```
  mpicc -o MiPrograma MiPrograma.c
  ```

  O bien utilizando el compilador C++:

  ```
  mpicxx -o MiPrograma MiPrograma.cpp
  ```

- Órdenes específicas para la ejecución de aplicaciones paralelas, normalmente denominada `mpirun` .
  Ejemplo:

  ```
  mpirun np 8 MiPrograma
  ```

- `MPI_Status` , una estructura que se rellena cada vez que se completa la recepción de un mensaje.
  Se pueden leer los campos de dicha estructura:
  - `status.MPI_SOURCE`: proceso fuente.
  - `status.MPI_TAG`: etiqueta del mensaje.

[1] no se especifica cómo se debe llevar a cabo la implementación de dicha interfaz para ninguna plataforma concreta

- Incluye también los tipos de datos bsicos que serán utilizados por los mensajes transmitidos 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: definen el contexto de comunicación de un grupo de procesos en MPI. Todas las funciones de comunicación necesitan como argumento un comunicador.
- Herramientas para monitorización y depuración de programas paralelos.

Como anteriormente se ha comentado, MPI está diseñado pensando en el desarrollo de aplicaciones SPMD. Este tipo de programas lanzan en paralelo N copias de un mismo programa, que se ejecutan por procesos asíncronos, es decir, hay que programar en el código de los procesos las instrucciones necesarias de semáforos para sincronizarlos. Dado que los procesos en MPI poseen un espacio de memoria completamente separado, el intercambio de información, así como la sincronización entre ellos, se ha de hacer exclusivamente mediante paso de mensajes.

3.4 Funciones MPI

En MPI se dispone de funciones de comunicación punto-a-punto para 2 procesos, así como funciones u operaciones colectivas para involucrar a un grupo de procesos. Si utilizamos MPI, entonces los nombres de todas las funciones han de comenzar con `MPI_`, la primera letra que sigue siempre es mayúscula, y el resto son minúsculas. La mayoría de las funciones MPI devuelven un entero, que hay que interpretarlo como un diagnóstico. Si el valor devuelto es `MPI_SUCCESS`, la función se ha realizado con éxito.

Los procesos pueden agruparse y formar *comunicadores*, lo que permite una definición del ámbito de las operaciones colectivas, así como un diseño modular.

El estándar MPI incluye funciones para realizar las siguientes operaciones:

- Básicas
- Comunicaciones punto–a–punto
- Comunicaciones colectivas
- Grupos de procesos
- Topologías de procesos
- Gestión e interrogación del entorno

3.4.1 Funciones básicas

MPI define un conjunto de funciones que permiten inicializar el entorno de ejecución de los procesos de una aplicación, y otras operaciones administrativas de la aplicación. Entre las funciones más utilizadas se encuentran:

- `MPI_Init`() para iniciar la ejecución paralela,
- `MPI_Comm_size`() para determinar el número de procesos que participan en la aplicación,
- `MPI_Comm_rank`(), para que cada proceso obtenga su identificador dentro de la colección de procesos que componen la aplicación,
- `MPI_Finalize`() para terminar la ejecución del programa.

A continuación se ve un ejemplo de programa que utiliza el *binding* de MPI para el lenguaje de programación C:

```
# include "mpi.h"
#include <iostream>
using namespace std;
main (int argc, char **argv) {
  int nproc; /*Numero de procesos */
  int yo; /* Mi direccion: 0<=yo<=(nproc-1)*/
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &yo);
  /* CUERPO DEL PROGRAMA */
  cout<<"Soy el proceso " <<yo<<" de "<<nproc<<endl;
  MPI_Finalize();
}
```

3.4.2 Comunicadores

Un comunicador es un elemento organizacional de MPI que define un grupo de procesos a los que se les permite comunicarse entre sí. Todas las funciones de comunicación de MPI necesitan como argumento una variable de tipo `MPI_Comm`, que en lo sucesivo llamaremos simplemente *comunicador*. Todo mensaje MPI debe especificar un comunicador a través de un nombre que se incluye en un parámtero explícito dentro de la lista de argumentos de una llamada MPI. El comunicador especificado en la llamadas a las operaciones de envío y recepción deben de concordar para que la comunicación pueda tener lugar.

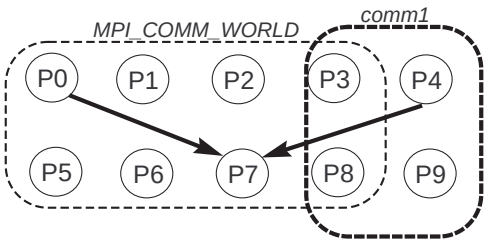


Figura 3.1: Representación gráfica de un comunicador.

Pueden existir muchos comunicadores, y un procesador específico puede ser miembro de de diferentes comunicadores. Dentro de cada comunicador, los procesadores están numerados

de forma consecutiva, comenzando en 0. Este número identificador se conoce con el nombre de `rank` del procesador en dicho comunicador. El `rank` también es utilizado para especificar el origen y el destino en las llamadas a las operaciones de envío y recepción. Si un procesador pertenece a más de 1 comunicador, su `rank` puede (y normalmente lo será) diferente en cada uno de ellos.

MPI automáticamente proporciona un comunicador básico denominado `MPI_COMM_WORLD`, que es el comunicador que comprende a todos los procesadores. Utilizando `MPI_COMM_WORLD`, cada uno de los procesadores puede comunicarse con cualquier otro. Se pueden definir comunicadores adicionales que consisten en subconjuntos de los procesadores disponibles.

Se habla de *contexto* de un comunicador como el ámbito de paso de mensajes en el que se comunica un grupo de procesos que pertenecen a un comunicador. Un mensaje enviado en 1 contexto sólo puede ser recibido en dicho contexto.

La identificación de los procesos participantes en 1 comunicador es unívoca:

- 1 proceso puede pertenecer a diferentes comunicadores
- Cada proceso tiene 1 identificador desde 0 a P-1.
  P es el número de procesos del comunicador.
- Mensajes destinados a diferentes contextos no interfieren entre sí

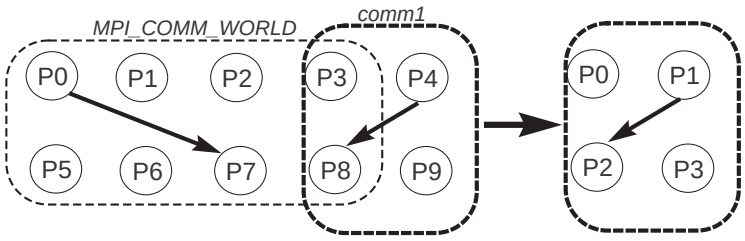


Figura 3.2: Comunicación dentro de un contexto

Cómo conseguir la información del comunicador

Un procesador puede determinar su `rank` dentro de un comunicador mediante una llamada a `MPI_COMM_RANK`. Cuando pensamos en `ranks`, debemos recordar que:

- Los `ranks` son consecutivos y comienzan en 0, tanto para C como para Fortran.
- Un procesador dado puede tener diferentes `ranks` en los diferentes procesadores a los que pertenece.

En la siguiente declaración de función de C `MPI_COMM_RANK`:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

El argumento `comm` es una variable de tipo `MPI_COMM`, un comunicador. Se podría utilizar `MPI_COMM_WORLD` aquí o, de forma alternativa, se podría dar el nombre de otro comunicador que se haya definido en cualquier otro lado. Dicha variable se declararía como sigue:

```
MPI_Comm some_comm;
```

Nótese que el segundo argumento es la dirección de la variable entera `rank`.

Tamaño de un comunicador

Un procesador puede también determinar el tamaño, esto es, el número de procesadores, de cualquier comunicador al que pertenezca con una llamada a `MPI_COMM_SIZE`. En la siguiente llamada de C a `MPI_COMM_SIZE`:

```
int MPI_Comm_size(MPI_Commcomm, int * size);
```

el argumento `comm` es del tipo `MPI_COMM`, un comunicador. El segundo argumento es la dirección de la variable entera tamaño.

Si el comunicador es `MPI_COMM_WORLD`, el número de procesadores devuelto por `MPI_COMM_SIZE` es igual al número definido desde:

- la línea de órdenes para `MPIRUN`. Por ejemplo:

  ```
  %mpirun -n 4 a.out
  ```

- Una variable de entorno dependiente del sistema (tal como `MP_PROCS`). El tamaño definido mediante esta variable de entorno es sobrescrito por el valor correspondiente que se pase a `MPIRUN`, si así lo queremos. Por ejemplo:

  ```
  % setenv MP_PROCS 4
  % a.out
  ```

3.4.3 Comunicaciones punto–a–punto

Se refiere a las comunicaciones "uno–a–uno" que se establecen entre un único par de procesos(ver figura 3.3). Es la operación de comunicación fundamental en MPI –la comunicación directa entre 2 procesadores, uno de los cuales envía datos y el otro recibe dichos datos. La comunicación punto–a–punto tiene 2 lados, lo cual significa que se necesitan 2 operaciones que han de programarse explícitamente: (a) envío y (b) recepción. Por tanto, los datos no se transfieren sin la participación de ambos procesadores.

Un mensaje, que contiene un bloque de datos transferido entre procesadores, consiste en un *envoltorio* que indica los procesadores origen y destino, así como un *cuerpo* que contiene los datos que realmente se van a transmitir. Los campos que componen el envoltorio y el cuerpo de los mensajes se pueden encontrar en los parámetros de las operaciones de enviar (`MPI_Send()`) y recibir (`MPI_Recv()`).

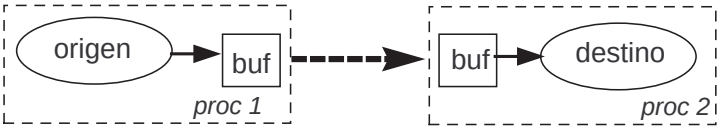


Figura 3.3: Operación básica de envío de mensaje

El *envoltorio* de 1 mensaje MPI tiene 4 partes (ver figuras 3.4, 3.5):

- *origen*: el proceso que envía;
- *destino*: el proceso que recibe;
- *comunicador*: especifica un grupo de procesos al cual pertenecen los procesos origen y destino en una comunicación;
- *etiqueta*: se utiliza para clasificar a los procesos. El campo etiqueta es obligatorio, pero su utilización se deja al programa. Un par de procesos comunicantes pueden utilizar valores etiqueta para distinguir entre diferentes clases de mensajes. Por ejemplo, podemos utilizar un determinado valor del campo *etiqueta* para discriminar a los mensajes que contengan datos y otro valor diferente para mensajes que contengan información de estado.

```
int MPI_Send (void *bufer, int cont, MPI_Datatype tipo_datos,
              int destino, int etiqueta, MPI_Comm comunicador)
```

Figura 3.4: Campos de una operación enviar con buffer

El *cuerpo* del mensaje MPI utiliza los 3 campos siguientes de información:

- *bufer*: la posición inicial de memoria donde se pueden encontrar (para la operación `Send`) los datos que se van a transmitir o la dirección de comienzo del array donde se van a almacenar los datos (para una operación `Recv`.
- *cont*: se refiere al número de elementos del tipo `tipodatos` que va a ser enviados.
- *tipo_datos*: el tipo de los datos que se van a enviar. En los casos más simples se trata de un tipo de datos primitivo, tal como `float`(*REAL*), `int` (*INTEGER*). En aplicaciones más avanzadas, este campo se puede ver como un tipo de datos definido por el usuario y construido a partir de los tipos de datos primitivos. En este caso se podrían considerar como análogos a `structure` del lenguaje C y pueden contener datos ubicados en cualquier lugar, esto es, no necesariamente en posiciones de memoria contiguas. Esta capacidad de poder hacer uso de tipos de datos definidos por el usuario nos permite tener una flexibilidad completa en la definición del contenido de los mensajes con MPI.

```
   int MPI_Recv (void *bufer, int cont, MPI_Datatype tipo_datos,
int origen, int etiqueta, MPI_Comm comunicador, MPI_Status *estado)
```

Figura 3.5: Campos de una operación recibir con buffer

Operaciones bloqueantes

La operaciones `MPI_Send()` y `MPI_Recv()` bloquean al proceso que las llama hasta que la operación de comunicación esté *completada*. Dicha situación se produce cuando bien el mensaje se copia en un buffer interno de MPI, o bien los procesos que envían y reciben sincronizan en el mensaje.

En la operacion `MPI_Send()` el cuerpo del mensaje contiene los datos que se van a enviar: unnúnero de elementos igual a `cont` del tipo `MPI_Datatype`. El envoltorio del mensaje indica dónse se van a enviar dichos datos. Además la llamada la función devuelve un código de error. La sintaxis concreta de la llamada a esta operación puede verse en la figura 3.4.

En la llamada a la operación `Recv()` los argumentos que pertenecen al envoltorio del mensaje determinan qué mensajes pueden ser recibidos cuando se produce la llamada. Los argumentos de la llamada a `Recv()`: `origen, etiqueta`, y `comunicador` han de concordar con los de la llamada a `Send()` del mensaje pendiente para que éste pueda ser recibido. Sólo se reciben los mensajes enviados desde el proceso `origen` que posean la `etiqueta` indicada en el argumento de la operación, aunque también se pueden utilizar argumentos comodín: `MPI_ANY_SOURCE , MPI_ANY_TAG`. No existen comodines para el argumento correspondiente al `comunicador`

Los argumentos de cuerpo de mensaje especifican dónde van a ser almacenados los datos que llegan, qué tipo se supone, y lo máximo que el proceso de recepción está dispuesto a aceptar. Si el mensaje recibido tiene más datos de los esperados, se produce un error y el programa se abortará. En general, el emisor y el receptor deben estar de acuerdo sobre el tipo de datos del mensaje, y es su responsabilidad garantizar dicho acuerdo. Si el emisor y el receptor utilizan tipos de datos incompatibles en los mensajes, los resultados serán indefinidos. El argumento `estado` devuelve información acerca del mensaje que se recibió. El origen y la etiqueta del mensaje recibido estarán disponibles de esta manera, que es la única si se utilizan comodines. También estará disponible el recuento real de los datos recibidos. Además, se devuelve un código que informa del resultado de la ejecución de la operación.

Con las operaciones de comunicación bloqueantes ninguna llamada a operación vuelve hasta que se completa. El concepto de *completación* es simple e intuitivo para la operación `Recv()`: ha llegado un mensaje que concuerda con los valores de los argumentso declarados en la llamada y los datos del mensaje han sido copiados en los argumentos–resultado de dicha llamada. En otras palabras, las variables pasadas a `MPI_Recv()` contienen 1 mensaje y están listas para ser utilizadas.

Para `MPI_SEND`, el significado de finalización es simple pero no es tan intuitivo. Una llamada a `MPI_SEND` se completa cuando el mensaje especificado en la llamada ha sido entregado al sistema MPI. En otras palabras, las variables pasadas a `MPI_SEND`, desde que se entregó el mensaje, pueden ser sobrescritas y reutilizadas. Recuérdese de la sección anterior que podría haber ocurrido una de estas dos cosas: (a) bien MPI copia el mensaje en un búfer interno para su entrega posterior, asíncrona, (b) o MPI esperó a que el proceso de destino recibiera el mensaje. Si MPI copia el mensaje en un búfer interno, entonces la llamada a `MPI_SEND` puede ser oficialmente completada, aunque el mensaje no haya salido aún del proceso de envío.

Si un mensaje pasado desde `MPI_SEND` es más grande que el búfer interno disponible en

MPI, entonces el proceso emisor debe bloquearse hasta que el proceso de destino comience a recibir el mensaje, o hasta que se disponga de más espacio de almacenamiento en el búfer. En general, los mensajes que se copian en el búfer interno de MPI ocuparán espacio de búfer hasta que comience el proceso de destino recibir el mensaje.

Una llamada a `MPI_RECV` coincide con un mensaje pendiente si coincide con la envoltura del mensaje pendiente (`fuente, etiqueta, comunicador`). También es necesaria la coincidencia de tipo de datos para la correcta ejecución pero MPI no la comprueba. En cambio, es obligación del programador el garantizar la coincidencia de los tipos de datos.

```
int main(int argc, char *argv[]) {
  int rank, size, mivalor, valor;
  MPI_Status estado;
  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, &estado );}
    else {
     MPI_Recv( &valor, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, &estado );
     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;
}
```

Figura 3.6: Mensajes de intercambio síncrono de datos entre pares de procesos

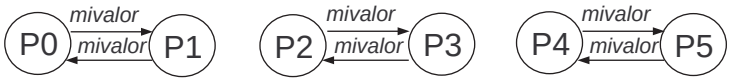


Figura 3.7: Representación gráfica de los intercambios.

Paso de mensajes síncrono

Se trata de una función de MPI que produce un envío sin buffer, bloqueante en las 2 partes de la comunicación. La función para envío síncrono `MPI_SSend()` (ver figura 3.6) presenta los mismos argumentos que la función `MPI_Send()`. La operación de envío finalizará sólo cuando la operación `Recv()` correspondiente sea invocada por otro proceso y el receptor haya comenzado a recibir el mensaje. Cuando la llamada a `MPI_Ssend()` vuelva, la zona de memoria que contiene el dato transmitido podrá ser reutilizada y el receptor habrá alcanzado el punto de su ejecución que corresponde a la llamada a la operación de recepción de mensajes.

Si la llamada de recepción del mensaje es `MPI_Recv()`, entonces la semántica del paso de mensajes es equivalente a 1 cita entre el emisor y el receptor.

Interbloqueos

En el programa de la figura 3.8, el proceso 0 intenta intercambiar mensajes con el proceso 1. Cada proceso comienza al intentar recibir un mensaje enviado por el otro; cada proceso se bloquea en espera de recepción. El proceso 0 no puede continuar hasta que el proceso 1 envía un mensaje; el proceso 1 no puede continuar hasta que el proceso 0 envía un mensaje. El programa es erróneo y se interbloquea. Nunca se enviará ningún mensaje, y tampoco nunca se recibirán mensajes. Se sabrá cuando un programa MPI está interbloqueado de la misma forma que se sabe que tenemos un bucle infinito en un programa secuencial: el programa está tardando demasiado tiempo para ejecutarse y nunca termina.

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
int myrank;
MPI_Status status;
double a[100], b[100];

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if( myrank == 0 ) {
    MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
    MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
 }
 else if( myrank == 1 ) {
   MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
   MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
 }
MPI_Finalize();
}
```

Figura 3.8: Código de 2 procesos que presenta interbloqueo

Objetos `status`

Aparecen en el último argumento de `Recv()` y también se pueden obtener llamando a la función que aparece en la figura 3.9. Permiten obtener información sobre el mensaje recibido. También permiten obtener el tamaño del mensaje recibido.

```
int MPI_Get_count (MPI_Status *estado,
                   MPI_Datatype tipo_datos, int cont)
```

Figura 3.9: Operación para obtener información del estado

Comunicación no bloqueante

Además del modo de comunicación bloqueante entre los procesos, en MPI se pueden utilizar operaciones de comunicación (`send` o `receive`) no–bloqueantes (ver figura 3.13).

```
int rank, size, flag, buf, src,tag;
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (rank == 0) {
  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
  for (int i=0; i<10; i++)
    MPI_Send( &buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD );
    ...
```

Figura 3.10: Sondeo continuo de varias fuentes emisoras desconocidas

Se puede conseguir una programación de aplicaciones distribuidas más flexible si se utilizan operaciones de comunicación no bloqueantes, ya que se evitan situaciones de interbloqueo debidas a la ordenación relativa de las operaciones de recepción y envío entre los procesos y que los procesos emisores bloqueen por enviar mensajes más largos que los buffers que van a contenerlos.

Las operaciones que vamos a estudiar son las siguientes:

- *Sondeo de mensajes.* Comunicación Asíncrona:
  - `MPI_Iprobe()` : Chequeo no bloqueante para un mensaje.
  - `MPI_Iprobe()`: Chequeo bloqueante para un mensaje.

La función `MPI_Iprobe()` no bloquea si no hay mensajes:

`MPI_Iprobe(int origen, int etiqueta, MPI_Comm comunicador, int *flag, MPI_Status *estado)`

Si hay mensaje pendiente, entonces hay que recibirlo con una llamada a `MPI_Recv`. Si como consecuencia de la llamada a dicha función: (flag > 0), eso indica que hay un mensaje pendiente que concuerda con los valores de los argumentos (origen, etiqueta, comunicador). El argumento pasado como parámetro en `MPI_Status` nos permite obtener más información sobre el mensaje pendiente de ser recibido. La llamada a la función `MPI_Probe()` retorna sólo cuando hay algún mensaje que concuerde con los argumentos (origen, etiqueta, comunicador):

`MPI_Probe(intorigen, intetiqueta, MPI_Commcomunicador, MPI_Status * estado)`

Permite esperar la llegada de un mensaje sin conocer su procedencia, etiqueta o tamaño.

```
int count, *buf, source;
// Me bloqueo hasta que se detecte un mensaje
MPI_Probe (MPI_ANY_SOURCE, 0,comm, &status);
// Se averigua el tama\~no y el proceso emisor del mensaje
MPI_Get_count(status, MPI_INT, &count);
source= status.MPI_SOURCE;
//Se reserva memoria para recibir el mensaje
buf=malloc(count*sizeof(int));
// Se recibe el mensaje
MPI_Recv(buf, count, MPI_INT, source, 0, comm, &status);
```

Figura 3.11: Recepción de mensaje con tamaño y fuente emisora desconocidos

- Envío-Recepción no bloqueantes sin buffer:
  - `MPI_Isend`: Inicia envío, pero retorna antes de copiar en el 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.

`MPI_Isend()` inicia una operación de envío de mensaje y vuelve inmediatamente, sin esperar a que se complete la llamada adicha función (ver 3.12). La completación de esta operación tiene el mismo significado que en 3.4.3, bien el mensaje se copia en un buffer interno de MPI, o bien los procesos emisores y receptores sincronizan en el paso de mensajes. Es importante hacer notar que las variables pasadas como argumentos en una llamada a `MPI_Isend` no pueden ser utilizadas –no deben ser ni siquiera leídas– hasta que la operación de envío invocada por la llamada haya completado.

```
int MPI_Isend(void* buf, int cont, MPI_Datatype tipo_datos, int destino
              int etiqueta, MPI_Comm comunicador, MPI_Request *request)
```

Figura 3.12: Operación MPI_Isend()

```
int MPI_Irecv(void* buf, int cont, MPI_Datatype tipo_datos, int origen
              int etiqueta, MPI_Comm comunicador, MPI_Request *request)
```

Figura 3.13: Operación `MPI_Irecv()`

El argumento `request` identifica la operación cuyo estado se pretende consultar o se espera que finalice. Se puede ver que no incluye el argumento `MPI_Status` como en el caso de la operación `MPI_Send()`. Con la función `MPI_Request_free(MPI_Request *request)` se puede liberar el objeto `request` de forma explícita.

El valor del argumento `MPI_Status` se puede obtener llamando a una de las funciones de chequeo de estado: `MPI_Test()`, `MPI_Wait()`. Es necesario realizar una llamada a estas funciones o a una de sus variantes para determinar el estado de completación cuando se programa con operaciones de comunicación no bloqueantes (`MPI_Irecv(), MPI_Isend()`). Si al volver

de la llamada la función `MPI_Test()` se tiene el argumento `flag >0`, entonces la función identificada por el argumento `request` ha finalizado, se libera a dicho objeto y se inicializa el objeto `estado`.

`MPI_Test(MPI_Request * request, int * flag, MPI_Status * estado)`

La operación `MPI_Wait()` produce bloqueo del proceso que la llama hasta que la operación chequeada finalice de forma segura.

`MPI_Wait(MPI_Request * request, MPI_Status * estado)`

Por último, es importante comentar que las operaciones bloqueantes: `MPI_Send(), MPI_Ssend(), MPI_Recv()` pueden ser conectadas con sus contrapartes no bloqueantes: `MPI_Isend()`, `MPI_Irecv()`.

3.5 Para ampliar

- Web oficial de OpenMPI: http://www.open-mpi.org
- Instalación de OpenMPI en Linux: http://www.underworldproject.org documentation/OpenMPIDownload.html
- Ayuda para las funciones de MPI: http://www.mpi-forum.org
- Tutorial de MPI: http://www.citutor.org

3.6 Ejercicios

3.6.1 Ejercicio 1: Algoritmo de ordenación paralelo

El algoritmo de ordenación denominado *pipesort* consiste en un encauzamiento (*pipeline*) de procesos paralelos en el que se entra una secuencia desordenada de items a ordenar. El resultado obtenido en la salida del encauzamiento es la secuencia ordenada.

En este ejercicio, cada proceso o *etapa* del encauzamiento tiene sitio para almacenar 1 carácter, que será el de orden lexicográfico mayor recibido hasta ese momento en dicha etapa.

En cada iteración de cualquier proceso se recibe 1 carácter por su *canal* de entrada, se compara con el almacenado anteriormente y el menor de los 2 es envíado a la siguiente etapa del encauzamiento mientras que el mayor de los 2 se almacena en la etapa actual. Supongamos que en el algún momento durante la ejecución del sistema, las 3 primeras etapas del encauzamiento tienen 1 carácter almacenado, mientras que la cuarta etapa no ha recibido todavía ningún caracter. Supongamos que el carácter 'b' es recibido en el canal de entrada del encauzamiento (ver Figura 3.14). El primer proceso compara el valor recibido con el valor almacenado, luego envía el menor de ambos ('b') a su canal de salida, y almacena el mayor de estos ('f'). De una forma similar, el segundo proceso, después de recibir el carácter ('b') enviado por el primer proceso, lo envía a su canal de salida y mantiene el carácter 'c' almacenado en su etapa. Sin embargo, la tercera etapa envía su carácter almacenado ('a') a su canal de salida y almacena

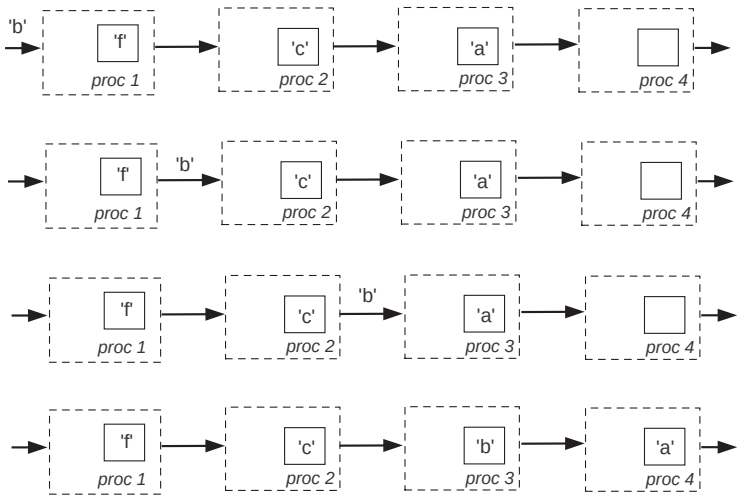


Figura 3.14: Ordenación de datos utilizando un encauzamiento.

el carácter que acaba de recibir ('b'). La cuarta etapa, que todavía no tiene ningún carácter, almacena el carácter recibido ('a') en su canal de entrada.

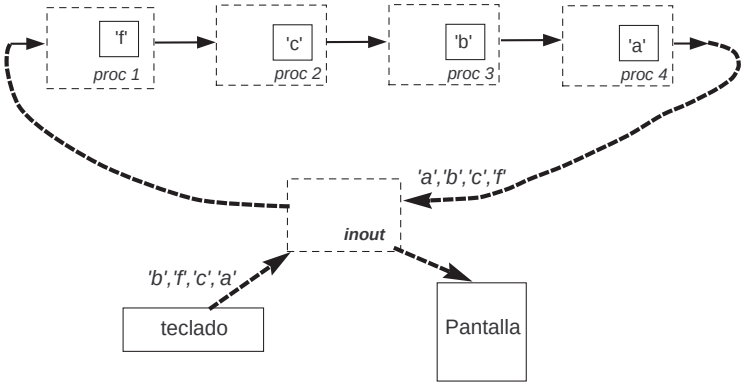


Figura 3.15: Proceso de entrada y salida al encauzamiento.

Será necesario programar un proceso genérico (Figura 3.15) que contenga el código de una etapa cualquiera del encauzamiento. Para poder construir el encauzamiento referido, las etapas del mismo se conectarán con el proceso **inout**, que lee la secuencia de caracteres del teclado, los envía de 1 en 1 al encauzamiento, recibe los caracteres de la secuencia ordenada y los muestra en la pantalla.

3.6.2 Ejercicio 2: Generación de N-primos

Se pretende programar una versión distribuida del famoso método denominado *criba de Erastótenes* para la obtención de los N primeros números primos.

Se utilizará una red de procesos concurrentes conectados mediante un pipeline, como el de la figura 3.16, para generar los N primeros números primos.

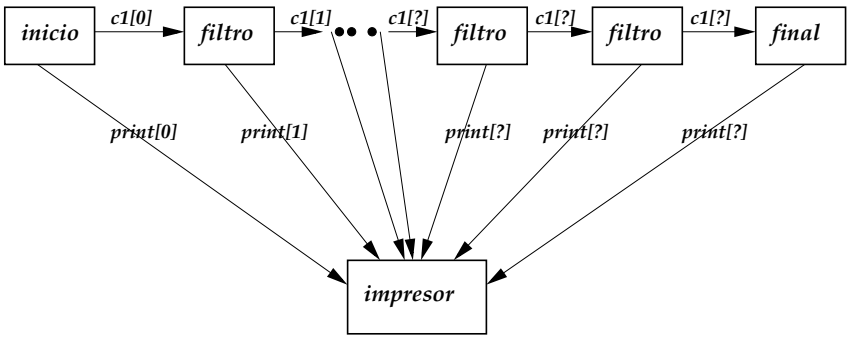


Figura 3.16: Encauzamiento de procesos que generan números primos.

Se conoce una constante *nLimite* que es mayor o igual que el N-ésimo número primo. El proceso *inicio* genera el primer número primo (el 2) y lo envía para que el proceso *impresor* lo imprima; después genera el siguiente número primo (el 3), lo envía al primer proceso filtro y continúa enviándole valores 5, 7, 11, ...

Cada proceso filtro comienza su ejecución recibiendo un valor primo del proceso anterior y enviándolo al proceso siguiente (según su posición en el pipeline); después recibirá contínuamente números, no necesariamente primos, desde el proceso anterior; si el número recibido es divisible por el valor primo que recibió al principio se descarta, si no, lo envía al proceso filtro siguiente. El proceso *final* recibe el último número primo, lo envía al proceso *impresor* y después descarta los demás números que reciba, tras lo cual termina.

El proceso *impresor* ha de ser programado de tal manera que pudiera recibir los valores primos de los N procesos del encauzamiento y los mostrase en pantalla según los va recibiendo.

3.6.3 Ejercicio 3: Colector de datos generados remotamente

Escribir un programa compuesto por un proceso servidor, denominado `colector()`, y `m` procesos `generadores()`, que imprime conjuntos de hasta `m` números enteros en distintas líneas de la pantalla. Los números enteros son generados aleatoriamente por los procesos de forma distribuida, enviados al proceso colector, imprimidos asíncronamente por cada proceso y, luego, imprimidos en el orden en que fueron recibidos por el colector. Se ha de procurar introducir cierto grado de no determinismo en la recepción de los números en el `colector`.

El proceso `colector ()` realizará un número de iteraciones $\mathtt{N} = (\mathtt{min}\{\mathtt{n}_1, \mathtt{n}_2, \ldots \mathtt{n}_\mathtt{m}\})^\mathtt{m}$, donde $\mathtt{n}_\mathtt{i}$, $\mathtt{i} = 1 \ldots \mathtt{m}$, son las iteraciones que realiza cada uno de los `m` procesos generadores. Inicialmente recibe el número de iteraciones que realiza cada proceso generador y calcula el número de iteraciones que él mismo ha de realizar. Para cada iteración espera hasta un tiempo T para recibir un mensaje con un número entero de cada uno de los procesos generadores. Después imprime todos los números recibidos en esa iteración. Cuando terminan las iteraciones del colector, entraría en la fase de terminación: recibe todos los mensajes pendientes de los procesos generadores, él mismo termina y, luego, terminará el programa completo.

Los procesos generadores obtienen un número entero en el rango `1 ... rangoIter` y se lo envían inicialmente al proceso colector. Este número indica las iteraciones que va a realizar el proceso generador. En cada iteración estos procesos tienen el siguiente comportamiento:

- Esperar un tiempo arbitrario.
- Generar un entero en el rango 1..rangoNum. Los números generados en cada iteración han de ser distintos.
- Enviar el entero anterior al colector.
- Imprimir el número generado y pasar a la siguiente iteración.

El proceso colector ha de programarse con un bucle que ha realizar `N` iteraciones (var arriba), antes de entrar en la fase de terminación.

3.6.4 Ejercicio 4: Juego de cartas

El ejercicio consiste en programar una simulación con procesos paralelos de un juego simple de cartas en el que en cada partida participan un número variable de jugadores y 1 *croupier*. Una partida se supone dividida en un número variable de *manos* (esta dato es fijo y hay que suministrarlo al programa). Cada jugador obtiene aleatoriamente una carta de 1 montón. Supondremos, por simplicidad, que el valor de cada carta se encuentra en el rango de `1 ... 5` puntos. Después de cogerla, se enseña la carta al *croupier* y el jugador hace una apuesta que implica, además de jugarse cierta cantidad de dinero, el hacer una estimación de la suma de los valores de las cartas que han retirado todos los jugadores participantes en esa mano. Los jugadores que acierten en su predicción, lógicamente, puede haber más de 1, obtendrán una cierta cantidad de puntos, de tal forma que aquellos jugadores que obtengan la máxima puntuación al final de la partida se quedarán con el dinero de los demás. Por tanto, el programa ha de presentar la información de los números de las manos en que ganó cada jugador.

Para resolverlo, vamos a suponer la siguiente actuación de los procesos:

- Jugadores: generan 2 números enteros (del 1 al 5) de manera aletoria. El primero de estos números se corresponderá con el valor de la carta y el segundo con el valor de la apuesta. Por supuesto, ambos números han de ser diferentes, esto es, si se repitieran en la misma mano, habría que volver a generar 1 de ellos. Después, se envían secuencialmente 3 mensajes al *croupier*: (1) el valor de la carta que han cogido para participar en la mano, (2) el valor numérico que constituye la predicción de la suma de puntos de las cartas de todos los participantes de esa mano, (3) un mensaje solicitando permiso para que el *croupier* pueda pasar a evaluar si el jugador ha ganado esa mano o no.
- Croupier: comprueba, en cada mano de la partida, si hay mensajes pendientes de ser recibidos de cada uno de los jugadores informando del valor de su carta. Si ya hay jugadores que han comunicado este valor, entonces este jugador pasa a ser *seleccionable*, es decir, si el `jugador[i]` comunicó el valor de su carta, entonces el proceso *croupier* convierte en seleccionable a ese jugador para que envíe el valor de su apuesta. Después,

se determinará entre los jugadores que hayan hecho una apuesta si han ganado la mano que actualmente se está jugando. Los mensajes que contengan el valor de la carta han de ser recibidos por el proceso croupier antes que los mensajes con las apuestas y los que envían los jugadores para que se determine si han ganado. Las comprobaciones de si hay alguno de estos mensajes ya enviado por los jugadores se hace iterativamente para cada una de las manos de una partida.

Por último, el método `main()` de la clase principal ha de generar la información, en forma de tabla, de los procesos que han ganado la partida actual, así como qué manos ganó cada uno de los jugadores.

3.6.5 Actividades a realizar

- Completar el código que falta en los procesos de los ejercicios seleccionados.
- Crear y lanzar la ejecución de los procesos de la clase `main()` de acuerdo con la *arquitectura* de comunicación de cada aplicación.
- ¿Existe alguna modificación para hacer alguno de los programas menos determinista? Si es así, explíquese dicha solución e intentar implementarla.
- ¿ Qué proceso sería el más indicado para decidir la terminación del programa completo en cada caso? Justificar la respuesta.

3.6.6 Documentación a entregar

Elaborar un archivo comprimido que incluya lo siguiente:

1. El código fuente completo de todos los programas que se hayan realizado.
2. Un archivo PDF que documente lo más importante de los programas que se hayan realizado e incluya también una salida significativa de cada programa.

Práctica 4

Programación de sistemas distribuidos usando RMI.

4.1 Resumen

El objetivo de esta práctica es familiarizarse con el uso de las principales tecnologías para el desarrollo de aplicaciones distribuidas [Liu, 2004], [Verissimo and Rodrigues, 2004]. Este guión se corresponde con el estudio y programación de aplicaciones que utilizan la interfaz RMI de Java [Hortsmann and Cornell, 2008], [Lea, 2001]. Se continúa con varios ejercicios adicionales de programación y se propone una plantilla para completar con cada uno de ellos.

4.2 Introducción

La tecnología estándar que propone Java para programación distribuida se denomina *Remote Method Invocation (RMI)* está basada en el modelo de llamadas remotas [Birrel and Nelson, 1984] y en la programación orientada a objetos, de ahí que se hable de invocación de métodos remotos , es decir, que pueden ser invocados desde el método `main()` de aplicaciones *cliente*.

RMI permite definir métodos remotos en las clases de Java. La información concreta acerca de qué métodos pueden ser llamados desde otros programas no se encuentra dentro de la clase, sino que se ha de incluir en una subinterfaz que hereda de la interfaz `Remote` (ver figura 4.1).

Utilizando RMI, en la parte del servidor se ha de programar una clase (ver figura 4.2) que implementa los métodos declarados en la interfaz remota. Es obligatorio declarar un método constructor [1] en dicha clase, ya que la claúsula que especifica tirar la excepción `RemoteException` es obligatoria.

Según se puede ver representado en la figura 4.3, antes que el servidor pueda comenzar a aceptar llamadas de los procesos clientes, se han de generar los archivos denominados *stubs* y *skeletons*.

[1] `sayHello(...)`, en el ejemplo presentado

```
import java.rmi.*;
public interface Hello extends Remote {
/**
 * Este metodo remoto
 * @devuelve un mensaje de tipo String.
*/
public String sayHello()
throws java.rmi.RemoteException;
} //fin interfaz remota
```

Figura 4.1: Interfaz de método remoto

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloImpl extends  UnicastRemoteObject
   implements Hello{

  public HelloImpl() throws RemoteException{
     super();
  }
  public String sayHello() throws RemoteException{
     returns ‘‘Hola a todo el mundo!!!’’;
  }
  //Programando el sirviente:
  public static void main(String args[]){
     try{
      String numeroPuerto = (br.readLine()).trim();
      String registroURL="rmi://localhost:"+numeroPuerto+"/hola";
      HelloImpl h = HelloImpl();
      Naming.rebind(registroURL, h);
      System.out.println(‘‘Servidor preparado.’’);
     }
     catch(RemoteException re){
        ...
     }
     catch (MalformedURLException e){
        ...
     }
  }
}
```

Figura 4.2: Implementación del servidor para el servicio `sayHello()`

Se generan automáticamente al compilar las clases con la orden `rmic` incluida en la distribución del entorno de desarrollo SDK de Java. Posteriormente habría que iniciar la ejecución

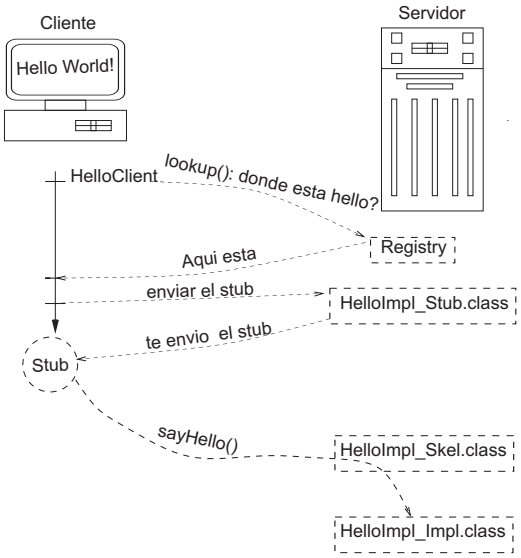


Figura 4.3: Representación de la ejecución de `HelloImpl.class`.

del servidor de nombres (*Registry*, en la figura) y arrancar el programa en la parte del servidor[2]. Cada *stub* contiene la signatura [3] de los métodos incluidos en la interfaz remota. Un *skeleton* tiene una función similar al *stub*, pero en la parte del servidor. En el ejemplo de la figura 4.2, los *stubs* y *skeletons* son generados a partir del código de la clase `HelloImpl.class` con la orden `rmic`.

Los procesos clientes, para poder utilizar el servicio del método `sayHello()`, han de programar en su código algo así como:

```
public static void main (String args[]){
   System.setSecurityManager(new RMISecurityManager());
  try{
      Hello h =
      (Hello) Naming.lookup(‘‘rmi://localhost:1050/hola’’);
      //se obtiene del registro la referencia del objeto remoto
      String mensaje = h.sayHello();
      System.out.println(‘‘ClienteHola:’’ + mensaje);
     }
  catch(remoteException re){
  ...
  }
}
```

Figura 4.4: Implementación en Java de un cliente del servicio `sayHello()`

[2]Ambas cosas se hacen de 1 sola vez con la orden: `java HelloImpl`

[3]nombre, lista y tipo de parámetros de una función o método

4.3 Ejercicio 1: Tutorial RMI

4.3.1 Definición de la interfaz

Este es el primer paso y consiste en definir las interfaces de los objetos con métodos que vayamos a invocar remotamente en nuestras aplicaciones:

```
import java.rmi.*;
public interface InterfazHola extends Remote {
/**
 * Este metodo remoto devuelve un mensaje.
 * @param  nombre - un String que contiene un nombre.
 * @devuelve un mensaje String.
*/
public String decirHola(String nombre)
throws java.rmi.RemoteException;
} //fin interfaz remota
```

Figura 4.5: Interfaz de método remoto

4.3.2 Implementación de la interfaz

La implementación de la `InterfazHola` hereda de la clase `Unicast RemoteObject`, lo que indica que la transmisión de la referencia del objeto se hará según un esquema de comunicación directa con los posibles clientes. Existe la opción de heredar de `MulticastRemoteObject`. La clase `ImplHola` ha de implementar 1 método.

En la figura 4.5 se describe un cierto tipo de objetos de Java cuyo método `decirHola(...)` devuelve un `String` y tira cualquier excepción remota, ya que si se producen han de ser tratadas en el contexto del objeto que llame al método.

Para completar la aplicación distribuida que sirve de ejemplo al tutorial que pretendemos desarrollar hay que programar, además de una implementación de la interfaz anterior, un objeto servidor y un cliente.

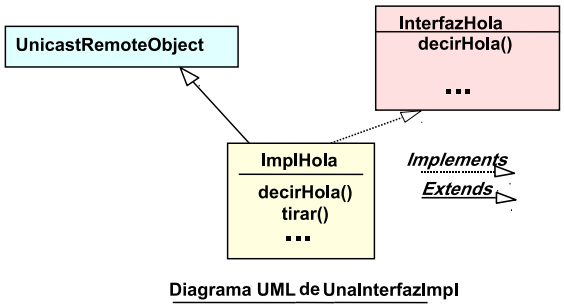


Figura 4.6: Representación de las relaciones de la clase `ImplHola`.

```
        import java.rmi.*;
        import java.rmi.server.*;
       / * Esta clase implementa la interfaz remota
        */
        public class ImplHola extends UnicastRemoteObject
            implements InterfazHola {

         public ImplHola() throws RemoteException {
             super( );
         }


         /* Programar la implementacion del metodo
          * public String decirHola(String nombre)
         */
} // fin clase implementacion de la interfaz Hola
```

4.3.3 Implementación del objeto servidor

La implementación del servidor se va a hacer con 2 clases, que pueden programarse dentro de un mismo archivo:

- La clase `ImplHola` que implementa los métodos remotos declarados en la interfaz `InterfazHola` anteriormente (sección 4.3.2).
- El *sirviente* que llamamos `ServidorHola`.

El objeto `ServidorHola` tiene las siguientes misiones:

a. Iniciar el *registro RMI de nombres* en la máquina servidora o crearlo si no hay ningún registro válido para el puerto que se indica en la instrucción:

`LocateRegistry.getRegistry(NumPuertoRMI)`

b. Crear una instancia del *servidor* para exportarlo al *registro RMI*:

`objExportado = new ImplHola()`

c. Registrar al objeto anterior en el servicio de nombres con el nombre simbólico "hola" y quedarse esperando a recibir llamadas de los clientes.

```
public class ServidorHola {
 public static void main(String args[]) {
  InputStreamReader is = new InputStreamReader(System.in);
  BufferedReader br = new BufferedReader(is);
  String numeroPuerto, registroURL;
```

```
    try{
      System.out.println("Entrar el numero de puerto del registro RMI:");
      numeroPuerto = (br.readLine()).trim();
      int numeroPuertoRMI = Integer.parseInt(numeroPuerto);
      lanzarRegistro(numeroPuertoRMI);
      //*** crear una instancia de la implementacion para exportarlo
      //*** la exportacion se ha de hacer utilizando rebind()
      //*** para ligarle una URL y un nombre simbolico, conocido en la red.

      System.out.println("El Servidor Hola esta preparado.");
     }
     catch (Exception re) {
            System.out.println("Excepcion en el main() del ServidorHola: " + re);
     }
    } // fin main
    private static void lanzarRegistro(int NumPuertoRMI) throws RemoteException{
      try {
           Registry registro = LocateRegistry.getRegistry(NumPuertoRMI);
           registro.list( ); // Esta llamada levantara una excepcion
      }
      catch (RemoteException e) {
        // No hay ningun registro valido en este puerto
        Registry registro = LocateRegistry.createRegistry(NumPuertoRMI);
      }
     // Este metodo lista los nombres registrados con el objeto registro RMI
     private static void listarRegistro(String registroURL)
            throws RemoteException, MalformedURLException {
        System.out.println("El Registro " + registroURL + " contiene: ");
        //*** El metodo Naming.list(registroURL); devuelve
        //*** una cadena que contiene todos los nombres registrados
        //*** programar dentro de un bucle la impresion de citados nombres.
     }
    } // fin class
```

4.3.4 Implementación del cliente

Encuentra la referencia del objeto remoto en el registro RMI, para lo cual hay que pasarle el nombre de *host* del servidor o su dirección `IP` [4] y el número de puerto donde se encuentra dicho registro. Se le asigna a un objeto de la interfaz remota (no de su implementación):

```
public class ClienteHola {
  public static void main(String args[]) {
     try {
      String nombreHost;
      InputStreamReader is = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(is);
```

[4] En Linux se obtiene con la orden `hostname -I`

```
        System.out.println("Entrar el nombre del host del Registro RMI:");
        nombreHost = br.readLine();
        System.out.println("Entrar el numero de puerto del registro RMI:");
        String numeroPuerto = br.readLine();
        //*** encontrar el objeto remoto (Naming.lookup(...)) y asignarlo
        // *** a un objeto de ``InterfazHola".
        System.out.println("Consulta completada " );
        //*** llamar al metodo remoto de igual modo que si fuera local

        System.out.println("ClienteHola: " + mensaje);
        } // fin try
       catch (Exception e) {
         System.out.println("Excepcion en el ClienteHola: " + e);
       }
    } //fin main
}//fin class
```

4.3.5 Ejecutar la aplicación en 1 sola máquina

Los pasos que hay que dar son los siguientes:

- Compilar con `javac ImplHola.java`
- Ejecutar la orden para el compilador: `rmic ImplHola`, asegurándose que estamos en el directorio donde se encuentra `ImplHola.java` y que `java\bin` se encuentra en nuestra ruta por defecto dentro de la variable de entorno `PATH`, así como la variable de entorno `CLASSPATH` está bien definida. Tras ejecutar esta orden se tendrán los archivos: `ImplHola.class, InterfazHola.class, e ImplHola_Stub.class`.
- Compilar el resto de los archivos Java de la aplicación: `ServidorHola, ClienteHola`, utilizando la orden: `javac *.java`.
- Lanzar la ejecución del servidor: `java ServidorHola` desde la línea de órdenes.
- Lanzar la ejecución del cliente: `java ClienteHola`.
- Por último, terminar (con Ctrl-C en DOS) o asesinar (kill en Linux) al servidor al terminar el ejercicio, para no dejar procesos *zombies* en el sistema.

4.4 Ejercicio 2: Implementación de un servicio de hora y fecha

En este ejercicio se trata de programar un servicio distribuido que proporcione el día de la fecha y la hora a los programas clientes, que se ejecutarán en máquinas diferentes de la que proporciona el servicio, y que lo invocan conforme a la declaración de método remoto de la siguiente interfaz:

```
// Un archivo de interfaz  RMI simple
import java.rmi.*;
/**
 * Esto es la interfaz remota del servicio
 */

public interface DiaDelaFechaInterface extends Remote {
/**
 * Este metodo remoto devuelve un mensaje.
 * @devuelve una fecha
 */
 public String getDaytime()
    throws java.rmi.RemoteException;
 } //fin interfaz remota
```

Figura 4.7: Interfaz del método remoto `getDaytime()`

4.4.1 Implementación del servidor

La implementación de `getDayTime()` declarado en la interfaz remota anterior es simplemente un *envoltorio* (tipo de *wrapper* de Java) del método correspondiente de la clase `Date()` del paquete `java.util.*`. El método devolverá la hora como una cadena (`String`), para lo cual se invocará al método `toString()` de la clase `Date()` anteriormente aludida.

```
/* Esta clase implementa la interfaz remota
 * DiaDelaFechaInterface
 */
 public class ImplFecha extends UnicastRemoteObject
     implements DiaDelaFechaInterface {

   //*** Programar aqui los metodos necesarios

 }
} // fin clase
```

Figura 4.8: Plantilla para implementación de la clase `ImplFecha`

4.4.2 Implementación del *sirviente*

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.net.*;
import java.io.*;
```

```
  /**
   * Esta clase representa al sirviente de un objeto
   * distribuido de la clase Fecha, que implementa la interfaz
   * remota DiaDelaFechaInterface.
   */

  public class ServidorFecha  {
   public static void main(String args[]) {
    //***Preparamos los objetos necesarios para leer de la corriente de
    //***entrada bloques de datos
     try{
    //Necesitamos saber el n\'umero de puerto bajo el que se va a ejecutar
    //el registro RMI
       System.out.println("Entrar el numero de puerto del registro RMI:");
    //*** leeremos el numeroPuerto

       lanzarRegistro(numeroPuertoRMI);

    // crear un objeto de la clase "ImplFecha" con el mensaje de la fecha
       /**********A completar***************/

    // construir el registroURL, para el protcolo rmi, concatenando en un
    //"string" el nombre de la m\'aquina servidora,
    // el numero de puerto y, al final, un nombre simb\'olico para el servicio
    //que estamos implementando como un objeto distribuido
    /**********A completar***************/

    // Ligar el registro RMI con la referencia anterior del objeto "implHola"
  /**********A completar***************/

       System.out.println("El Servidor Fecha esta preparado.");
    }// fin try
    catch (Exception re) {
       System.out.println("Excepcion en el main() del  ServidorFecha: " + re);
    } // fin catch
} // fin main

 // Este metodo inicia un registro RMI en el "host" local, si
 // todavia no existe en el numero de puerto especificado.
 private static void lanzarRegistro(int NumPuertoRMI)
    throws RemoteException{
    try {
       //obtener una referencia de objetos del registro RMI para que el
       //servicio dee nombres se ejecute bajo el puerto especificado
       // en el argumento
       /**********A completar***************/
      // La siguiente llamada levantara una excepcion si el registro RMI
      // no existe todavia
       registro.list();
    }
```

```
     catch (RemoteException e) {
     //Si no hay ningun registro RMI valido ejecutandose en el numero que
     //se paso como argumento, entonces hay que crear un registro
     //en el puerto indicado
        /**********A completar***************/
     }
  } // fin lanzarRegistro
 // Este metodo lista los nombres registrados con el objeto registro RMI
    private static void listarRegistro(String registroURL)
    throws RemoteException, MalformedURLException {
        /**********A completar***************/
    } //fin listarRegistro
  } // fin class
```

4.4.3 Implementación de un programa de prueba

El cliente, por último, acaba imprimiendo el resultado:

```
import java.io.*;
import java.rmi.*;
/**
 * Esta clase representa al objeto cliente de un objeto distribuido
 * de la clase Fecha, que implementa la interfaz remota
 * DiaDelaFechaInterface
 */
public class ClienteFecha {
  public static void main(String args[]) {
     try {
      //Preparamos los objetos necesarios para leer de la corriente de
      //entrada bloques de datos.
     // Ahora vamos a necesitar leer el nombre de la m\'aquina en la que
     //se ejecuta el objeto servido, asi como el numero de puerto bajo
     //el que se ejecuta el Registro RMI
     int PuertoRMI;
     String nombreHost;
     InputStreamReader is = new InputStreamReader(System.in);
     BufferedReader br = new BufferedReader(is);
     System.out.println("Entrar el nombre del host del Registro RMI:");
     nombreHost = br.readLine();
     System.out.println("Entrar el numero de puerto del registro RMI:");
     String numeroPuerto = br.readLine();
     PuertoRMI = Integer.parseInt(numeroPuerto);
     // construir el registroURL, para el protcolo rmi, concatenando en
     //un "string" el nombre de la m\'aquina servidora, el numero de puerto y,
     //al final, el nombre simbolico del servicio que se implementa con del
     //objeto distribuido
    /**************A completar************************
     // encontrar el objeto remoto (lookup) en servicio de nombres (Naming)
```

```
         // del Rgistro RMI  y transmitirlo a un objeto de la interfaz
         // remota ("InterfazHola")
        /**************A completar************************/
         System.out.println("Consulta completada " );
         // Ahora, ya poddemos llamar al metodo remoto y mostrarlo en la
         //patalla local
       /**************A completar************************/
      } // fin try
      catch (Exception e) {
         System.out.println("Excepcion en el ClienteFecha: " + e);
      }
   } //fin main
}//fin class
```

4.4.4 Configuración distribuida de la aplicación

La aplicación que hemos desarrollado se ha de configurar para que se ejecute en una máquina diferente de la que alberga la implementación de las clases del servidor (ServidorFecha, ImplFecha).

En esta configuración se necesitará que en el directorio de la máquina cliente se encuentre la interfaz remota `DiaDelaFechaInterface` y el resguardo del cliente `ImplFecha_Stub.class`, junto con el fuente del cliente, para que se pueda compilar éste último. Lo demás es igual al caso de configurarlo todo en 1 máquina, salvo que habrá que proporcionarle al cliente el nombre de *host* (o su dirección IP) del servidor y el puerto a través del que se accede al servicio de registro RMI. Los pasos para ejecutar una aplicación en 2 máquinas diferentes son los siguientes:

a. Seguir los pasos indicados en la sección 4.3.5

b. En la máquina cliente ejecutar:

```
myhost:/USER/Docencia/SCD/practicas/practica4> java ClienteFecha
Entrar el nombre del host del Registro RMI:  192.168.1.36
Entrar el numero de puerto del registro RMI: 1050
Consulta completada
Fri Jan 06 12:56:49 CET 2012
```

4.5 Ejercicio 3: Implementación de un servicio de consultas

Utilizando RMI, escribir una aplicación que puede servir de prototipo de un sistema más elaborado de consultas de opinión. El sistema podría utilizarse, por ejemplo, para pasar encuestas a los alumnos de una determinada asignatura. Suponer que sólo se va a encuestar sobre 1 tema. Los entrevistados pueden escoger: *sí, no, ns\nc.*

Escribir una aplicación servidora, que acepte los votos, guarde el recuento en memoria y proporcione el recuento actual a aquellos usuarios del servicio que estén interesados en conocer los resultados. El servicio que se va a programar ofrecerá a sus usuarios los métodos incluidos en la siguiente interfaz:

```
import java.rmi.*;
/**
 * Esto es una interfaz remota para consultar una encuesta.
 * Se definen las signaturas de los metodos enviarRecuento,
 * aceptarVoto y guardarRecuento
 */
public interface InterfazConsultas extends Remote {
   public String enviarRecuento() throws java.rmi.RemoteException;

   public String aceptarVoto(int voto) throws java.rmi.RemoteException;

   public void guardarRecuento(int voto) throws java.rmi.RemoteException;
} //fin interfaz remota
```

4.5.1 Implementación del servidor

```
import java.rmi.*;
import java.rmi.server.*;
/**
 * Esta clase implementa la interfaz remota
 * InterfazConsultas
 */
public class ImplConsultas extends UnicastRemoteObject
     implements InterfazConsultas {

   //*** Crear el array donde se guardara el recuento de
   //**cada una de  las 3 posibles respuestas
   //*** Programar el constructor de la clase
   /**
   * Comprueba que el voto es correcto y lo acepta
   * @param voto: numero que indica la votacion del cliente
   * Si es correcto se llama a guardarRecuento(voto) y
   * se devuelve un mensaje informando.
   * si no, tambien se informara con un mensaje.
   */
   public String aceptarVoto(int voto) throws RemoteException{
   /******A Completar ***********************************/
   }
   /**
   * Una vez comprobado que el voto es correcto lo contabiliza
   * en nuestro array
   */
```

```
   public void guardarRecuento(int voto) throws RemoteException{
   /***** A completar *****************/
   }
   /**
   * Devuelve al cliente el array con el recuento realizado
   */
   public String enviarRecuento() throws RemoteException{
        return "Si:"+\\dato del array de recuentos +"  No:"
\\dato del array de recuentos+"  ns/nc:"+\\dato del array de recuentos;

   }
} // fin clase
```

4.5.2 Implementación del sirviente

```
   import java.rmi.*;
   import java.rmi.server.*;
   import java.rmi.registry.Registry;
   import java.rmi.registry.LocateRegistry;
   import java.net.*;
   import java.io.*;
   /**
    * Esta clase representa al objeto servidor de un objeto
    * distribuido de la clase Consultas, que implementa la interfaz
    * remota InterfazConsultas
    */
   public class ServidorConsulta  {
    public static void main(String args[]) {
       //***Preparamos los objetos necesarios para leer de la corriente de
      //***entrada bloques de datos
       try{
      //Necesitamos saber el n\'umero de puerto bajo el que se va a ejecutar
      //el registro RMI
         System.out.println("Entrar el numero de puerto del registro RMI:");
      //*** leeremos el numeroPuerto

         lanzarRegistro(numeroPuertoRMI);

      // crear un objeto de la clase "ImplConsultas" desde el que se
      //ejecutaran los metodos
         /**********A completar***************/
      // construir el registroURL, para el protcolo rmi, concatenando en un
      //"string" el nombre de la m\'aquina servidora,
      // el numero de puerto y un nombre simb\'olico para el servicio
      //que estamos implementando como un objeto distribuido
      /**********A completar***************/

      // Ligar el registro anterior con la referencia anterior del objeto
```

```
      //"implConsultas"
    /**********A completar***************/
         System.out.println("El Servidor Consultas esta preparado.");
      }// fin try
      catch (Exception re) {
         System.out.println("Excepcion en el main() del ServidorConsultas: " + re);
      } // fin catch
  } // fin main
   // Este metodo inicia un registro RMI en el "host" local, si
   // todavia no existe en el numero de puerto especificado.
   private static void lanzarRegistro(int NumPuertoRMI)
      throws RemoteException{
      try {
         //obtener una referencia de objetos del registro para que el
         //servicio dee nombres se ejecute bajo el puerto especificado
         //en el argumento
         /**********A completar***************/
        // La siguiente llamada levantara una excepcion si el registro
        //no existe todavia
         registro.list();
      }
      catch (RemoteException e) {
      //Si no hay ning\'un registro v\'alido ejecut\'andose en el n\'umero que se
       //paso como argumento, hay que crear un registro en el puerto indicado
         /**********A completar***************/
        System.out.println(
           "Registro RMI creado en el puerto " + NumPuertoRMI);
      }
   } // fin lanzarRegistro
  // Este metodo lista los nombres registrados con el objeto registro
  private static void listarRegistro(String registroURL)
     throws RemoteException, MalformedURLException {
       System.out.println("El Registro " + registroURL + " contiene: ");
       String [ ] nombres = Naming.list(registroURL);
       for (int i=0; i < nombres.length; i++)
          System.out.println(nombres[i]);
  } //fin listarRegistro
} // fin class
```

4.5.3 Implementación de un programa de prueba

```
   import java.io.*;
   import java.rmi.*;
   /**
    * Esta clase representa al objeto cliente de un objeto distribuido
    * de la clase Consultas, que implementa la interfaz remota
    * InterfazConsultas
    */
```

```
public class ClienteConsulta {
  public static void main(String args[]) {

   try {
      //Preparamos los objetos necesarios para leer de la corriente
      //de entrada bloques de datos.
      // Ahora vamos a necesitar leer el nombre de la m\'aquina en la que
      // se ejecuta el objeto servidor, asi como el numero de puerto desde
      //el que se ejecuta el Registro RMI.

      /**************A completar ***************************/
      // Tambien leemos la contestacion que da el cliente a una pregunta.
      System.out.println("Cree que el Granada debe hacer un esfuerzo
                para mantener a Uche?");
      System.out.println("Conteste con si, no o ns/nc:");
      String voto = br.readLine();
      // Asignamos un valor a decision erroneo
      // cambiamos ese valor si la respuesta se
      // ha realizado correctamente y asignamos a cada
      // contestacion su numero de decision correspondiente
      /******************A completar ***********************/

      // construir el registroURL, para el protcolo rmi, concatenando
      //en un "string" el nombre de la m\'aquina servidora, el numero de puerto
      //y, al final, el nombre simb\'olico del
     //servicio que se implementa con del objeto distribuido
     /**************A completar************************/

      // encontrar el objeto remoto (lookup) en servicio de nombres (Naming)
      //del Registro RMI y transmitirlo a un objeto de la
      //interfaz remota ("InterfazConsultas")
     /**************A completar************************/

     System.out.println("Consulta completada " );
      // Ahora, ya podemos llamar al metodo remoto que se encarga de aceptar
      // el voto emitido y mostrarlo en la patalla local
    /**************A completar************************/

     //Leemos otra linea por pantalla para preguntar al cliente si quiere
     //ver el recuento obtenido hasta ahora
System.out.println("Quiere ver el recuento obtenido hasta ahora? (si o no):");
     String recuento = br.readLine();

     // Llamamos al metodo remoto si la contestacion ha sido afirmativa
     if(recuento.equalsIgnoreCase("si")){
/************* A completar **************/

         System.out.println(mensaje);
     }
   } // fin try
```

```
    catch (Exception e) {
       System.out.println("Excepcion en el ClienteConsulta: " + e);
    }
   } //fin main
  }//fin class
```

4.6 Documentación a entregar

Elaborar un archivo comprimido que incluya lo siguiente:

a. El código fuente completo de todas los programas que se hayan realizado.

b. Un archivo PDF que documente lo más importante de los programas que se hayan realizado e incluya también una salida significativa de cada programa.

c. Comentar cualquier incidencia que se haya producido en el desarrollo de esta práctica (p.e.: problemas con la configuración distribuida) y cómo se han resuelto finalmente.

d. Opcionalmente, desarrollar algún ejemplo de aplicación distribuida con RMI utilizando las plantillas de los ejercicios anteriores.

Práctica 5

Programación de tareas periódicas con prioridades usando hebras POSIX.

5.1 Resumen

Esta práctica se engloba dentro del marco teórico de planificacion RMS/RMA(Rate Monotonic Scheduling/Analysis) [Burns, 2003], [Wellings, 2004]. RMS/RMA comprende un conjunto de métodos de planificación con asignación estática de prioridades a las tareas más frecuentes de un programa concurrente y de tiempo real, que proporciona una base para el diseño de sistemas de tiempo real muy complejos. A cada tarea del programa se le asigna una única prioridad que se establece en razón inversa a su periodo,es decir, cuanto menor sea el periodo de cada tarea (mayor frecuencia), ésta obtendrá una mayor prioridad a la hora de ser planificada por el sistema: $\forall\, T_i,\ T_i < T_j \ \Rightarrow\ P_i > P_j$.

5.2 Introducción

Los tests de planificabilidad permiten predecir a-priori si un conjunto de procesos es planificable o no. Entre ellos, el más ampliamente utilizado es que se deriva del *Teorema de Liu & Layland*(1972), que establece una condición suficiente de planificabilidad para un conjunto de tareas de tiempo–real.

Teorema 1 *En un sistema de n tareas periódicas independientes, con prioridades asignadas en orden de frecuencia, se cumplen todos los plazos de respuesta, para cualquier desfase inicial de las tareas, si el factor de utilización del procesador (U) cumple la siguiente desigualdad:* $U = \Sigma_{i=1}^{n} \frac{C_i}{T_i} \leq 2 \cdot (2^{1/n} - 1) = U_0(n)$.

Donde C_i es el tiempo de ejecución en el peor de los casos (o *worst-case execution time*, WCET) de la tarea i-ésima; T_i se refiere al periodo de dicha tarea; $\frac{C_i}{T_i}$ es el factor de utilización del procesador, por parte de esa tarea T_i; U es el factor de utilización total del sistema por parte

del conjunto de n-tareas $\{T_i\}$; y $U_0(n)$ es la utilización máxima del procesador que garantiza la planificación de las n tareas, p.e., para n=3 tiene el valor .779 (ver figura 5.1) sin que ninguna de ellas pierda ningún tiempo límite durante toda su ejecución.

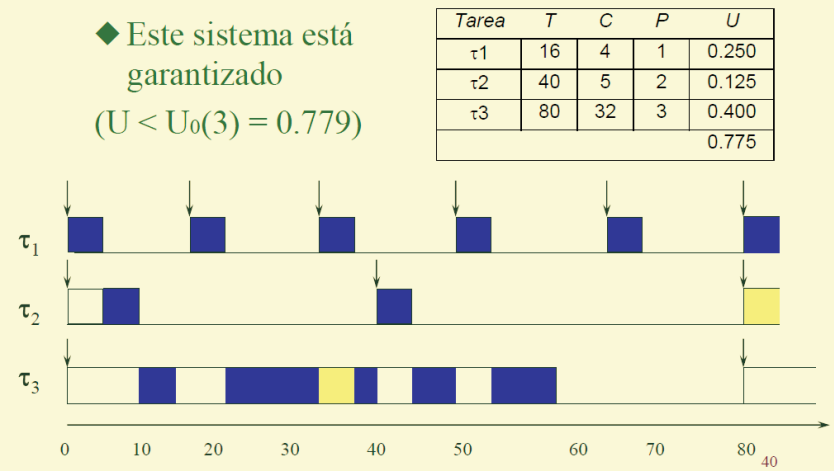


Figura 5.1: Ejemplo de diagrama de Gantt de tres tareas planificables por teorema de Teorema Liu & Layland

5.3 Rapitime

RapiTime(c) es un conjunto de herramientas que permiten analizar el comportamiento temporal de un software empotrado en una aplicación más general. Se utiliza principalmente para optimización, depuración y validación de requerimientos relacionados con el tiempo de ejecución de un software que tenga cumplir con requerimientos de tiempo real estrictos. El aspecto fundamental que interesa en esta práctica es el hecho de que RapiTime(c) puede determinar el tiempo de ejecución en el peor de los casos (WCET) de un determinado módulo–software en una aplicación qur tenga partes críticas duranet su ejecución En el *reportviewer* que proporciona la herramienta RapiTime(c) además se puede observar un informe que indica cómo han contribuido las diferentes módulos o funciones que componen dicho software y, de esta manera, podemos realizar dicho cálculo. En la figura 5.2 se puede observar un ejemplo de un informe generado para la aplicación de ejemplo `hello.c` que se suministra con este software:

5.3.1 Manejo de la herramienta

Para facilitar la evaluación de cualquier módulo o función–software se proporciona la plantilla identificada con el nombre `my_application.c`:

```
/*=================================================================
* Rapitime
* Template, which can be reproduced and modified to form new projects
* Application code
*
* (c) 2006 Rapita Systems Ltd. All rights reserved
```

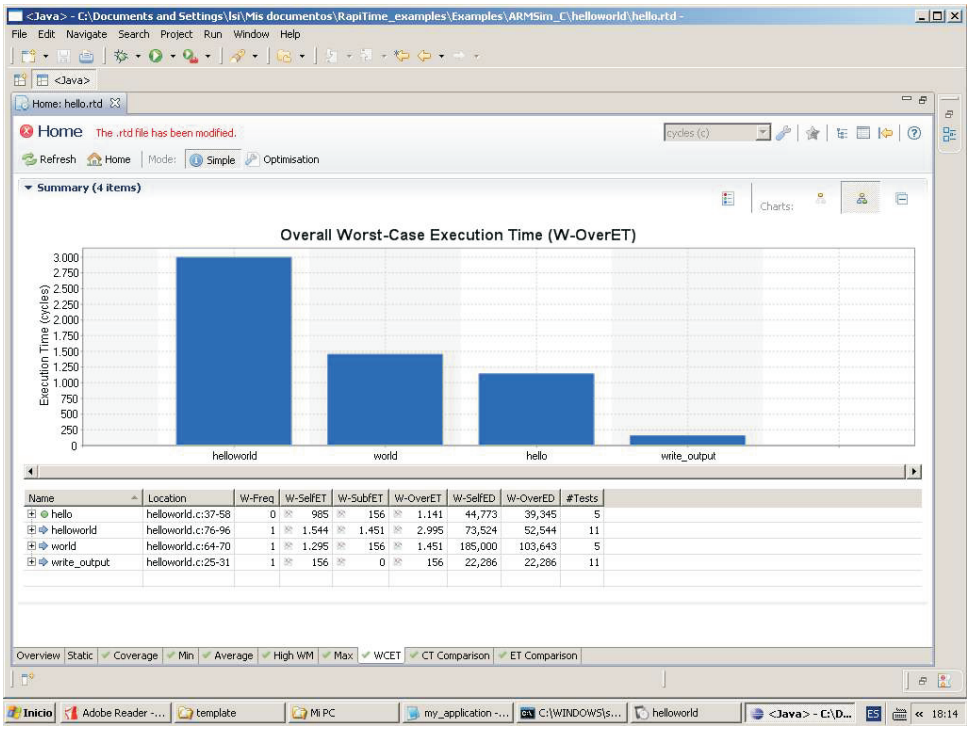


Figura 5.2: WCET para el programa de ejemplo helloworld.c en RapiTime

```
*====================================================================
*
* CPU : MCP555
* TRACE : Software buffer, on chip timer method
* EXTRACTION : BDM debugger
* TARGET : phycorempc555
* Compiler : ppc-gcc-elf
*
*====================================================================
*/
#include "rpt.h"
#include "my_application.h"
/*
* Test code analysis root function. The name of this function has to
* be set as ROOT in Makefile
*
* To illustrate how to put together a RapiTime test project, we have
* taken the count_set_bits routine from the msg_handler example.
*/
```

Código a sustituir en la plantilla:

```
/*---------------------------------------------------------*/
/* REPLACE THIS WITH YOUR TEST CODE */
/*---------------------------------------------------------*/
/*******************************************
* count_set_bits
* counts the number of set bits in the bit
* pattern supplied
*******************************************/
Uint8
count_set_bits( Uint8 value )
{
Uint8 set_bits = 0;
/* While value contains uncounted, set bits */
while( value )
{
if( value & 0x1 )
{
/* Found a bit, count it */
set_bits++;
}
value >>= 1;
}
/* All bits have been counted, return the result */
return set_bits;
}
```

```
/*
* Test harness. The objective is to call the code to be tested with
* representative inputs so as to achieve an extensive range of
* different run-time behaviours.
*
* In this case, we exhaustively test count_set_bits from the
* msg_handler example.
*/
/* Do not instrument the test fail trap */
#pragma RPT instrument( "test_failed", FALSE );
void
test_failed( void )
{
/*
* In the event of a functional failure, the code will loop
* here. When interactively debugging, a breakpoint can be
* placed here.
*/
```

```
for( ;; ) ;
}
void
my_main_function( void )
{
```

Código a sustituir en la plantilla:

```
/*-------------------------------------------------------------*/
/* REPLACE THIS WITH YOUR TEST HARNESS */
/*-------------------------------------------------------------*/
Uint8 mask;
Uint16 count;
for( count = 0; count <= 31255; count++ )
{
/* Create 8-bit mask from low bits of 16-bit count */
mask = count;
/* Call count_set_bits with every different possible input value
* and check the result against an independent implementation.
*/
if( count_set_bits( mask ) != ( ( mask >> 7)
+ ( ( mask >> 6 ) & 0x1 )
+ ( ( mask >> 5 ) & 0x1 )
+ ( ( mask >> 4 ) & 0x1 )
+ ( ( mask >> 3 ) & 0x1 )
+ ( ( mask >> 2 ) & 0x1 )
+ ( ( mask >> 1 ) & 0x1 )
+ ( mask & 0x1 ) ) )
{
test_failed( );
}
}
/*-------------------------------------------------------------*/
```

```
}
```

En dicho código habrá que sustituir la parte incluida en las cajas por el código que se quiera evaluar. En el caso de la práctica, por el de cada una de las tareas del programa. Luego, hay que ejecutar `go` y posteriormente `make`, que se encontrarán en el mismo directorio de la plantilla. Como resultado en el *reportviewer* se podrán ver los informes correspondientes con el WCET buscado (ver figura 5.3).

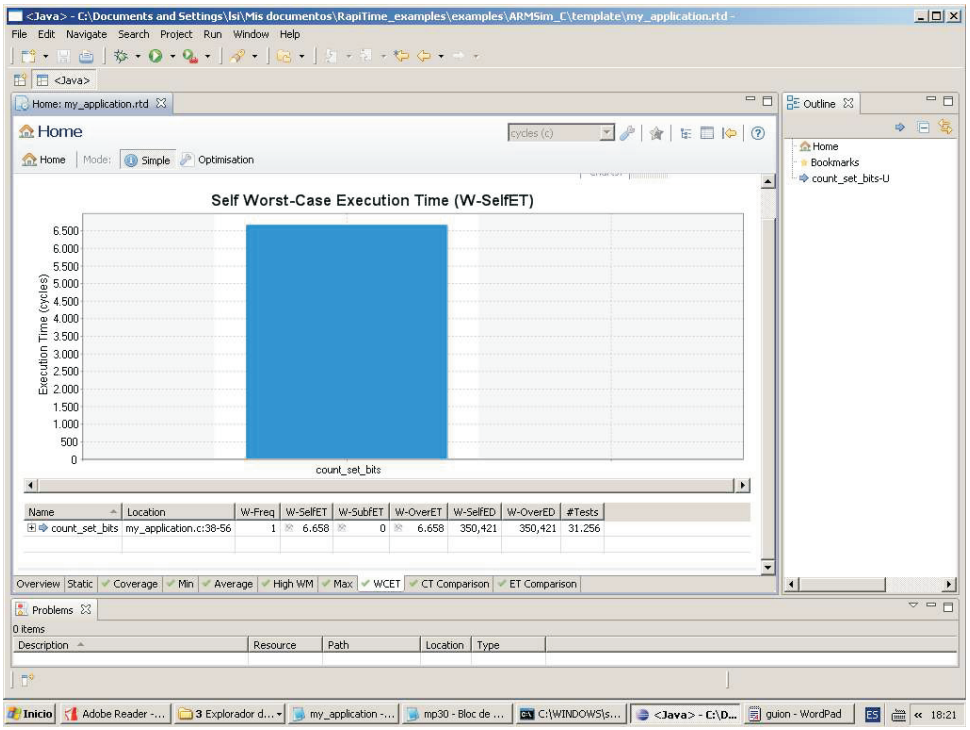


Figura 5.3: WCET para el programa de my_application.c en RapiTime.

5.4 Software de simulación de planificación de tareas en tiempo real

De cara a la realización de la práctica, se facilita un sencillo programa en `C` que simula la planificación de tres tareas periódicas. Este programa, que usa `pthreads` (threads POSIX), funciona con el núcleo `2.6.31-11-rt` de Ubuntu para tiempo real. Hay que tener en cuenta que la versión de Ubuntu-rt incluye una interfaz muy recortada de funciones que normalmente se utilizan en programación de tiempo real. Por ejemplo, no se pueden utilizar temporizadores `"setitimer()"`, ya que conduce al bloqueo del programa y esto no está documentado en ningún sitio. La versión del programa de las tres tareas que se muestra a continuación ha sido programada sólo con funciones RT–POSIX 2001, a saber:

- `pthread_sigmask(...)`: para bloquear las señales incluidas dentro de un conjunto (o máscara de señales), en lugar de utilizar `sigprocmask(...)`, ya que esta llamada no funciona con `pthreads`.

- se ha programado utilizando las funciones: `timer_create(..)` y `timer_settime(...)`, en lugar de `setitimer(...)`, ya que ésta tampoco funciona (la función no vuelve nunca) y da lugar a que comience a ejecutarse la rutina de interrupción indefinidamente.

- Si se utilizan las nuevas funciones (`timer_settime(..)`, etc.), hay que cambiar la representación interna de los datos referidos al tiempo. Hay que utilizar una estructura `timespec`, que incluye nanosegundos, en lugar de la estructura `timeval` que sólo llega hasta los microsegundos. Esto implica cambiar la función para obtener el tiempo del reloj de tiempo real. Es necesario ahora utilizar una nueva función para obtener el tiempo: `clock_gettime(...)`, en lugar de `gettimeofday(...)`.

El programa funciona si el número de iteraciones de las tareas no es muy alto. Otra cosa es que las iteraciones para cada tarea hay que añadirlas "a mano", modificando el código fuente.

```
#include "mis_entradas.h"
#include <signal.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>
#include <malloc.h>
#include <limits.h>
#include <string.h>
#include <sys/types.h>
#include <sys/signal.h>
#include <sched.h>
#define TAREA1     (SIGRTMAX-2) //Mas prioritaria
#define TAREA2 (SIGRTMAX-1)
#define TAREA3 (SIGRTMAX)
#define MILLON 1000000

timer_t SetTimer(int signo, struct timespec t, int mode);
void manejador_senial(int signo, siginfo_t * info, void *context);
volatile int flag=0;

static void funcion_plan(void* p);
timer_t id_tarea1, id_tarea2, id_tarea3;
char *getopt_flags="n:1:2:3:";
char *prognombre;
tipo_params *parametros;
static void *conversion(void *p){
  tipo_params *p0= (tipo_params*)p;
  p0->intervalo.it_interval.tv_nsec= (NANO_SEGS_DEF/1000)*atoi(optarg);
  p0->intervalo.it_interval.tv_sec=0;
  while( p0->intervalo.it_interval.tv_nsec >= 1000000){
     p0->intervalo.it_interval.tv_nsec -= 1000000;
     p0->intervalo.it_interval.tv_sec++;
    }
  return NULL;
  }

void uso(){
 printf( "Uso correcto: %s {-n numero iter. tarea de prueba}
                           {-1 periodo tarea1 (x10msegs.)}
                           {-2 periodo tarea2}
                           {-3 periodo tarea3} \n", prognombre);
 }
void manejador_senial(int signo, siginfo_t * info, void *context)
{   int i;
if (signo == TAREA1) {
      long iter1= parametros[0].num_iter;
      float suma;
```

```
        clock_gettime(CLOCK_REALTIME, &parametros[0].tiempo_comienzo);
        for (i=1;i<iter1*1000;i++){ //Simula la carga de la tarea
            suma = suma + 1.0/iter1;
            }
        clock_gettime(CLOCK_REALTIME, &parametros[0].tiempo_fin);
        }
        else if (signo == TAREA2) {
         long iter2= parametros[1].num_iter;
         float suma;
         clock_gettime(CLOCK_REALTIME,&parametros[1].tiempo_comienzo);
         for (i=1;i<iter2*1000;i++){
            suma = suma + 1.0/iter2;
         }
         clock_gettime(CLOCK_REALTIME, &parametros[1].tiempo_fin);
        }
        else if (signo== TAREA3){
          float suma;
          long iter3= parametros[2].num_iter;
          clock_gettime(CLOCK_REALTIME, &parametros[2].tiempo_comienzo);
          for (i=1;i<iter3*1000;i++){
            suma = suma + 1.0/iter3;
          }
          clock_gettime(CLOCK_REALTIME, &parametros[2].tiempo_fin);
          }
        else if (signo == SIGINT) {
            timer_delete(id_tarea1);
                timer_delete(id_tarea2);
                timer_delete(id_tarea3);
                perror("Ctrl + C capturado!\n");
                exit(1);
        }
}
static void funcion_plan(void* p){
   long n_iter;
   // int i;
   tipo_params *p0 = (tipo_params*)p;
   sigset_t mask;
   n_iter= p0->num_iter;
   // siginfo_t si;
   //Bloquear al temporizador temporalmente
   sigemptyset(&mask);
   if (p0->num_t==0)sigaddset(&mask, TAREA1);
   else if (p0->num_t==1)sigaddset(&mask, TAREA2);
   else if (p0->num_t==2)sigaddset(&mask, TAREA3);
   else perror("Error en mascara\n");

 if (pthread_sigmask(SIG_BLOCK, &mask, NULL) == -1){
          perror("Ha fallado sigprocmask");
   }
    if (p0->num_t==0)  id_tarea1= SetTimer(TAREA1, p0->intervalo.it_interval, 1);
```

```
    else if (p0->num_t==1) id_tarea2= SetTimer(TAREA2,  p0->intervalo.it_interval, 1);
    else if(p0->num_t==2)id_tarea3= SetTimer(TAREA3,  p0->intervalo.it_interval, 1);
    else perror("Error en armar temporizadores\n");
   //Desbloquear temporizadores
   if (pthread_sigmask(SIG_UNBLOCK, &mask, NULL)==-1){
      perror("Ha fallado sigprocmask desbloqueado las seniales\n");
    }
   // sleep(20);
   return;
}
timer_t SetTimer(int signo, struct timespec t, int mode){
  // El primer argumento sera el numero de interrupcion
  // El segundo un valor en milisegundos
  // El tercero el modo== (1|0) : 1== temporizador periodico ;
  //                              0== temporizador de 1 solo disparo
    struct sigevent sigev;
    timer_t timerid;
    struct itimerspec itval;
    struct itimerspec oitval;
    // Crearse un temporizador POSIX para generar la interrupcion numero == "signo"
    sigev.sigev_notify = SIGEV_SIGNAL;
    sigev.sigev_signo = signo;
    sigev.sigev_value.sival_ptr = &timerid;
    if (timer_create(CLOCK_REALTIME, &sigev, &timerid) == 0) {
      itval.it_value.tv_sec = t.tv_sec;
      //Lo convierto en nanosegundos
      itval.it_value.tv_nsec = (long)(t.tv_nsec) * (1000L);
      if (mode == 1) {
         //Para crear un temporizador
    itval.it_interval.tv_sec = itval.it_value.tv_sec;
    itval.it_interval.tv_nsec =  itval.it_value.tv_nsec;
      } else { //Para crear un temporizador de un solo disparo
          itval.it_interval.tv_sec = 0;
          itval.it_interval.tv_nsec = 0;
       }
      //Armar el temporizador
      if (timer_settime(timerid, 0, &itval, &oitval) != 0) {
          perror("Error en la llamada a time_settime!");
           }
     } else {
        perror("Error en la creacion del temporizador\n!");
             }
     return timerid;
}
int main(int argc, char **argv){
   #define MAXLINEA 20
   //char linea1[5]="comp";
   //char linea2[5]="plan";
   //char linea3[5]="nada";
   //char linea_e[MAXLINEA];
```

```
 pthread_t *h;
 pthread_attr_t *atr;
 struct sched_param *param_planif;
 int n_hebras=3, i=0, c;
 long e_total[3],  n_iter[3];
 prognombre= argv[0];
 h= (pthread_t*) malloc(sizeof(pthread_t)*n_hebras);
 atr= (pthread_attr_t*) malloc(sizeof(pthread_attr_t)*n_hebras);
 parametros= (tipo_params*) malloc(sizeof(tipo_params)*n_hebras);
 param_planif= (struct sched_param*) malloc(sizeof(struct sched_param)*n_hebras);
  //Asignacion peligrosa
 //  parametros= &paramtrs[0];
  /*****Pero necesaria para que se comunique con la funcion manejador_senial()*/
 struct sigaction sigact;
 sigemptyset(&sigact.sa_mask);
 sigact.sa_flags = SA_SIGINFO;
 sigact.sa_sigaction = manejador_senial;
 //Asignar "sigaction" para capturar la senial
 if (sigaction(TAREA1, &sigact, NULL) == -1) {
    perror("Ha fallado sigaction\n");
     return -1;
  }
 if (sigaction(TAREA2, &sigact, NULL) == -1) {
    perror("Ha fallado sigaction\n");
    return -1;
  }
  if (sigaction(TAREA3, &sigact, NULL) == -1) {
    perror("Ha fallado sigaction\n");
    return -1;
  }
 //Montar un manejador para capturar la senial CTRL+C
 //y utilizarlo para salir del programa
 sigaction(SIGINT, &sigact, NULL);
   if (argc <5){ uso(); exit(1);}
while((c=getopt(argc, argv, getopt_flags))!= -1) switch(c){
case 'n':
  for ( i=0; i<n_hebras; i++){
   parametros[i].num_iter = atoi(optarg);
   parametros[i].num_t=i;
   n_iter[i]= parametros[i].num_iter;
  }
  // printf("\nIteraciones tarea de prueba: H0->%ld \n", n_iter[i]);
   break;
case '1':
     if (! conversion(&parametros[0]))
     printf("Periodo de la tarea H-0: %ld seg. %ld000 nseg.\n",
         (long) parametros[0].intervalo.it_interval.tv_sec,
        (long) (parametros[0].intervalo.it_interval.tv_nsec));
  break;
case '2':
```

```
   if (! conversion(&parametros[1]))
         printf("Periodo de la tarea H-1: %ld seg. %ld000 nseg.\n",
          (long) parametros[1].intervalo.it_interval.tv_sec,
        (long) (parametros[1].intervalo.it_interval.tv_nsec));
   break;
 case '3':
    if (! conversion(&parametros[2]))
    printf("Periodo de la tarea H-2: %ld seg. %ld000 nseg.\n",
          (long) parametros[2].intervalo.it_interval.tv_sec,
       (long) (parametros[2].intervalo.it_interval.tv_nsec));
    break;
   default:
     uso();
   exit(1);
 }
 if (parametros[0].num_iter==0){
   printf("Necesito el numero de iteraciones(espera activa)-> ");
   scanf("%ld",&n_iter[0]);
   parametros[0].num_iter= n_iter[0];
  printf("->%ld\n", parametros[0].num_iter);
 }
  while(i<n_hebras){
  if(pthread_attr_init(&atr[i]))
     fprintf(stderr,"Error inicializando parametros\n");
  if (pthread_attr_setstacksize(&atr[i], PTHREAD_STACK_MIN + MY_STACK_SIZE))
      fprintf(stderr, "error al asignar la pila de la hebra-0 \n");
  if (pthread_attr_setinheritsched(&atr[i], PTHREAD_EXPLICIT_SCHED))
     fprintf(stderr, "error al decir como se fija la politica de
                     planificacion de la hebra-0 \n");
  if (pthread_attr_setschedpolicy(&atr[i], SCHED_FIFO))
      fprintf(stderr, "error al fijar la politica de
                       planificacion de la hebra-0 \n");
  param_planif[i].sched_priority= (sched_get_priority_max(SCHED_FIFO)-i);
  if (sched_setscheduler(i, SCHED_FIFO, &param_planif[i]) < 0)
      fprintf(stderr, "sched_setscheduler\n");
  i++;
  }
  i=0;
  while(i<n_hebras){
  if (pthread_create(&h[i], &atr[0], (void*)funcion_plan, &parametros[i]))
      fprintf(stderr, "error al crear la hebra-0 \n");
  i++;
  }
  sleep(1);
  i=0;
  while(i<n_hebras){
    if (pthread_join(h[i], NULL))
     fprintf(stderr, "error en el join de la hebra-0\n");
    else{
        e_total[i]= MILLON*(
```

```
                    parametros[i].tiempo_fin.tv_sec- parametros[i].tiempo_comienzo.tv_sec)
                    +(parametros[i].tiempo_fin.tv_nsec-parametros[i].tiempo_comienzo.tv_nsec)
                    /1000;
          printf("Tiempo finalizacion de H-%d ->%ld microsegundos;
                  Num. de iteraciones %ld \n", i,e_total[i], parametros[i].num_iter);
      }
      i++;
    }
    //  funcion_plan(&parametros[0]);
    return 0;
}
```

El archivo de cabecera: `mis_entradas` es el siguiente:

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>   // needed for getrusage
#include <sys/param.h>
#define MAXLINEA 20
#define PRE_ALLOCATION_SIZE (100*1024*1024) /* 100MB pagefault free buffer */
#define MY_STACK_SIZE       (100*1024)      /* 100 kB is enough for now. */
#define N_ITER 3
#define SECS_DEF 0
#define MICRO_SEGS_DEF (1000000/HZ)  /*HZ es una constante del sistema
                                       y significa el tamanio de "tick" del reloj*/
#define NANO_SEGS_DEF (1000000000/HZ)
#define USEC_PER_SEC    1000000L
#define INST_CRITICO 2
#define PASADO_T_LIMITE -2147483647
typedef struct{
  struct itimerspec intervalo;
  struct timespec tiempo_comienzo;
  struct timespec tiempo_fin;
  long num_iter;
  int num_t;
} tipo_params;
int consigue_linea(char s[], int lim)
 {
   int c, i;
   i=0;
   while(lim-->0 && (c=getchar())!=EOF && c!='\n')
     s[i++]=c;
   if (c=='\n')
     s[i++]=c;
   s[i]='\0';
   if ((i==0)|| s[0]=='\n') return(0);
   else return(i);
 }
```

```
int  igual(char s[], char t[]){
int i;
int r;

for (i=0; s[i] !='\0' && s[i]==t[i]; i++);

if ((s[i] == '\0')&&(t[i]=='\0' || t[i]=='\n'))
  r=1;
else r=0;

return r;
}
```

5.5 Ejercicio

El ejercicio a realizar consiste en programar tres tareas que ejecutan funciones sencillas, que realizan operaciones sobre los elementos de un vector. Estas tareas se podrían englobar dentro del contexto de un controlador de parámetros leídos por una serie de sensores situados en unas instalaciones que precisen monitorización continua, por ejemplo, de los valores de humedad, temperatura, radiación, etc. El número de operaciones que realice cada función dependerá directamente del número de elementos del vector, que vendría determinado por el número de sensores activos. El tamaño máximo del vector será 200, número que se ha elegido para obtener con facilidad los tiempos de ejecución en el peor caso de las tareas aludidas. La primera tarea multiplica todos los elementos del vector por un valor constante y comprueba que se encuentren dentro de un determinado rango después de aplicarles la constante. También rellena otro vector del mismo tamaño, indicando si el elemento correspondiente se encuentra dentro del rango establecido (0) o no lo está (1). Devuelve 1 si hay al menos un elemento fuera del rango.

La segunda tarea ordena el vector de menor a mayor utilizando el método de la burbuja.

La tercera tarea busca la posición del vector a partir de la posición en que la suma de los elementos siguientes es menor que la suma de los elementos anteriores y ese elemento. Esta función se podría aplicar al vector ordenado para determinar un posible reparto de suministros.

5.5.1 Obtención de los tiempos de ejecución de peor caso

Para obtener los C_i de cada una de las tareas, se utilizarán las plantillas que se comentaron en la sección 5.3.1. Se pueden incluir en el mismo proyecto del tutorial de RapiTime(c), incluyendo estos archivos en `C` y modificando el `Makefile` para cada caso, lo que se indica en la línea del ROOT. El tamaño mayor del vector para las funciones puede tomarse alrededor de 200, ya que con valores mucho mayores, el tiempo que emplea el programa para calcular los peores tiempos de ejecución puede ser demasiado grande y llegar incluso a bloquearse. Indicar en la configuración de la herramienta que se muestren los resultados en milisegundos y especificar la frecuencia de reloj del ordenador utilizado (3,4 GHz suele ser un valor normal).

5.5.2 Aplicación del algoritmo de Liu & Layland

Para aplicar los tests de planificabilidad, se ha de redondear los valores de los tiempos de ejecución de peor caso obtenidos con la herramienta. A continuación, se fijaraán los periodos asignados a cada tarea, de forma que se cumpla la condición suficiente de planificabilidad del algoritmo RM. Hay que tener en cuenta para realizar correctamente este apartado que el factor de utilización del sistema (U) no puede superar el límite para tres tareas $U_0 = 0.779$. Realizar una tabla en la que se reflejen los valores siguientes: T_i, C_i, P_i y U_i para cada una de las tres tareas.

5.5.3 Optimización del factor de utilización del sistema

Aunque la planificabilidad de las tareas anteriores está asegurada con los periodos dados en el apartado anterior 5.5.2, sin embargo, es posible reducir el tiempo de dichos periodos para conseguir un factor de utilización del sistema mayor, sin que estas tareas dejen de ser planificables. Comprobar esto experimentalmente, es decir, utilizar el armazón–software dado por el programa C en la sección 5.4 para mostrar que se pueden encontrar nuevos valores de los periodos con los que se puede conseguir un factor que se aproxime al 99%, sin que ninguna de las tareas pierda algún tiempo límite.

5.6 Documentación a entregar

Elaborar un archivo comprimido que incluya lo siguiente:

a. El código fuente completo de todas los programas que se hayan realizado.

b. Un archivo PDF que documente lo más importante de los programas que se hayan realizado e incluya también una salida significativa de cada programa.

c. Comentar cualquier incidencia que se haya producido en el desarrollo de esta práctica (p.e. configuración del núcleo rt-Ubuntu, respuesta inesperada de las funciones de este sistema, etc.) y cómo se han resuelto finalmente.

d. Opcionalmente, desarrollar algún ejemplo de optimización del factor de utilización para varias tareas de tiempo real utilizando las plantillas anteriores.

Lista de Figuras

Bibliografía

[Birrel and Nelson, 1984] Birrel, A. and Nelson, B. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59.

[Burns, 2003] Burns, A. (2003). *Sistemas de tiempo real y lenguajes de programación*. Addison-Wesley/Pearson Education, Madrid.

[Darwin, 2004] Darwin, I. (2004). *Curso de Java*. O'Reilly-Anaya, Spain.

[Eckel, 2009] Eckel, B. (2009). *Piensa en Java (4a edicion)*. Prentice Hall, Spain.

[Gallmeister, 1995] Gallmeister, B. (1995). *Programming for the real world: POSIX 4.0*. O'Reilly, Sebastopol, California.

[Hortsmann and Cornell, 2008] Hortsmann, C. and Cornell, G. (2008). *Core Java, Vol. 2: Advanced Features (8th Edition)*, volume II of *Java Series*. The Sun Microsystems Press, USA.

[Hortsmann and Cornell, 2012] Hortsmann, C. and Cornell, G. (2012). *Core Java Volume I– Fundamentals (9th Edition)*, volume I of *Java Series*. The Sun Microsystems Press, USA.

[Lea, 2001] Lea, D. (2001). *Programación concurrente en Java: principios y patrones de diseño*. Addison-Wesley/Pearson Education.

[Liu, 2004] Liu, M. (2004). *Computación distribuida: fundamentos y aplicaciones*. Pearson Education, Madrid.

[Sánchez-Allende, 2005] Sánchez-Allende, J. (2005). *Java 2. Iniciación y referencia (2a ed)*. MacGraw-Hill, Spain.

[Snir et al., 1999] Snir, M., Otto, S., Huss-Lederman, S., and D. Walker, J. D. (1999). *MPI: The Complete Reference*. The MIT Press, Cambridge, USA.

[Verissimo and Rodrigues, 2004] Verissimo, P. and Rodrigues, L. (2004). *Distributed systems for system architects*, volume I. Kluwer Academic, N.Y. (USA).

[Wellings, 2004] Wellings, A. (2004). *Concurrent and real–time programming in Java.* John Wiley, N.J. (USA).

Indice Alfabético