Notas de Clase
Curso Infraestructura de Cómputo – ISIS 2203
Departamento de Ingeniería de Sistemas y Computación
Universidad de los Andes

Harold Castro
Sandra Rueda

Enero 2022

Índice

Introducción

Este documento está diseñado para ofrecer una guía básica para abordar los cuatro módulos del curso Infraestructura de Cómputo: Concurrencia, Virtualización, Seguridad y Planeación de Capacidad. En todos los capítulos hay actividades propuestas para revisar la comprensión del tema. Además, animamos al lector a profundizar y buscar información adicional.

1 Concurrencia

1.1 Introducción

Este capítulo presenta una introducción a la concurrencia. Primero entenderemos de qué se trata la concurrencia y qué problemas genera, para luego si buscar soluciones.

Concurrencia: coincidencia, concurso simultáneo de varias circunstancias

Estamos hablando de cosas que suceden al tiempo y esa simultaneidad puede generar algunos problemas. En este curso, esas cosas son los programas que se ejecutan en un computador. La concurrencia que nos ocupa es la ejecución simultánea de varios programas o aplicaciones en un mismo computador. Existen varias razones para querer ejecutar varias aplicaciones al mismo tiempo en un computador, siendo las principales:

1. Facilitar el modelo de tareas independientes pero simultáneas. Esto permite que una tarea compleja se subdivida en varias subtareas con un mejor encapsulamiento, lo que redunda en una mejor separación de los problemas. La programación Orientada Objeto es un ejemplo donde este concepto se aplica de manera extensiva porque cada clase modela de manera independiente la estructura y comportamiento de cada componente de la solución, haciendo explícita la relación entre ellos.
2. Mejorar la reactividad del sistema. Dada la capacidad actual de los computadores, es normal que queramos hacer varias cosas al mismo tiempo con ellos. Mientras utilizamos una aplicación, esperamos que otras (u otros módulos de la misma aplicación) sigan avanzando al mismo tiempo. Y si estamos interactuando con una aplicación cualquiera, esperamos respuestas inmediatas.
3. Aumentar el desempeño de un computador. Cuando tenemos aplicaciones muy pesadas que incluyen varios componentes, esperamos que, si la aplicación lo permite, se vaya avanzando en diferentes aspectos de la ejecución al mismo tiempo, para así disminuir el tiempo que debe esperar un usuario para obtener su respuesta.
4. Responder mejor a las necesidades de los ambientes modernos. Las interfaces gráficas, la red de datos y en general todas las tareas simultáneas que esperamos poder ejecutar, requieren de un adecuado manejo de la concurrencia por parte de los sistemas modernos.

Actividad 1-1: Para cada una de las razones anteriormente expuestas, identifique una aplicación o situación que la aproveche o la pueda aprovechar

Hoy en día los computadores son tan veloces que es lógico pensar que hagan varias cosas mientras nosotros hacemos solo una. Por ejemplo, mientras usted lee este texto (usando para ello una aplicación como Acrobat), el computador tiene la capacidad de hacer otras cosas (seguramente usted mismo está ejecutando Spotify en este momento, recibiendo un correo o descargando una película de la cual ya verificó que tiene los derechos de autor que lo permiten, etc.). Es esa velocidad del computador, y el manejo adecuado de la concurrencia, la que permite que los programas se estén ejecutando al mismo tiempo y usted no *sienta* interrupciones en ninguno de ellos. Aquí la palabra clave es sentir, no quiere decir que eventualmente no haya interrupciones, pero lo importante es que no nos demos cuenta de ellas.

La concurrencia es un modelo y no una característica del hardware. Esto significa que modelamos las soluciones de manera concurrente, luego las ejecutamos en el hardware disponible (que puede o no tener capacidades que

favorezcan la concurrencia). Si el hardware cuenta, por ejemplo, con varios procesadores (paralelismo), observaremos que varias tareas (tantas como procesadores tengamos) estarán efectivamente ejecutándose en el mismo instante. Sin embargo, si el número de tareas es superior al de procesadores, algún tipo de estrategia para compartir esos procesadores será necesaria para asegurar una buena ilusión.

En el resto del documento, revisaremos los conceptos de sincronización, procesos, programación asincrónica y servidores concurrentes que son parte de los elementos a estudiar sobre la concurrencia. A lo largo de la presentación usaremos Java, C o ejemplos en pseudo-código para ilustrar los diferentes mecanismos ofrecidos por los sistemas operativos.

El contenido de este documento está basado en tres fuentes principales: “Modern Operating Systems” de Andrew Tanenbaum, “Operating Systems: design and concepts” de Abraham Silberschatz, Peter Galvin y Greg Gagne y “Operating Systems: Three Easy Pieces” de Remzi y Andrea Arpaci-Dusseau.

1.2 Programas en ejecución

Para ejecutar un programa necesitamos disponer del código (secuencia de instrucciones) y de los datos que ese código manipula. Llamaremos proceso (o thread, indistintamente por ahora) a la ejecución secuencial de las instrucciones manipulando los datos, es decir, **un proceso es un programa en ejecución**. Este simple modelo da lugar a varias posibilidades: que un mismo código sea utilizado por varios procesos sobre conjuntos de datos diferentes, lo que daría lugar a varias instancias de un mismo programa; que un mismo código sea utilizado por varios procesos sobre el mismo conjunto de datos, lo que daría lugar a un programa multiproceso (o multithread); que unos mismos datos sean utilizados por varios procesos (que ejecutan códigos distintos), habilitando la posibilidad de compartir datos; o que un proceso utilice varios conjuntos de datos, habilitando la distribución o replicación de datos.

Con lo anterior tenemos entonces que, en una máquina en un momento dado, puede haber varios programas en ejecución (procesos) que utilizan conjuntos de código y conjuntos de datos de la manera que consideren adecuada. La ejecución de estos procesos tiene lugar en el procesador, este es el único elemento de un computador capaz de ejecutar las instrucciones que vienen en el código. Pero ¿cómo hacer si el número de procesos supera el número de procesadores? Los primeros computadores no tenían opción, tocaba terminar un proceso primero para poder ejecutar un segundo. Las máquinas tenían un único procesador y tenían que esperar la ejecución de un programa, que podía tardar horas, para poder ejecutar otro que solo tomaba unos minutos; o peor aún, si el programa en ejecución necesitaba un dato de un dispositivo externo (algo tremendamente lento con respecto a la velocidad del procesador), la CPU quedaba bloqueada porque nadie la podía utilizar. A esto sistemas se les conoce como **sistemas batch** y aún hoy día se utilizan en contextos muy específicos.

Para 1960 era evidente que esta ejecución secuencial de los programas era inconveniente. Apareció entonces el concepto de **multiprogramación**. La idea de la multiprogramación es que, si tengo varios procesos por ejecutar, vaya avanzando con todos “*al tiempo*”. Al tiempo tiene un significado especial en este contexto, significa que ejecuto un poco un proceso, luego paso al siguiente, luego a otro y en algún momento vuelvo al primero y continúo desde donde lo dejé. Aquí es importante la granularidad con la que miremos un sistema. Si miro con un lente grueso efectivamente varios procesos estarán ejecutándose al tiempo, pero si lo hago con un lente muy fino, podría decir que en un instante solo se encuentra un proceso en ejecución, aquel que tiene el procesador en ese instante.

Dos retos se destacan en los sistemas de multiprogramación: por un lado, se debe guardar el estado de un proceso para poder volver más tarde y continuar con la operación desde el último punto donde quedó (no queremos repetir trabajo ya avanzado), y por otro, los programas deben aceptar ser interrumpidos en cualquier momento sin que se afecten sus resultados. Lo segundo tiene que ver con una característica conocida como la **apropiación** (*preemption*) del procesador por parte del sistema operativo. Se llama sistema apropiativo a un sistema operativo que, sin la cooperación del proceso afectado, es capaz de quitarle el procesador a un proceso con la intención de continuar su ejecución en un momento posterior.

La multiprogramación dio lugar a los sistemas de **tiempo compartido**, donde varios usuarios comparten una misma máquina con un solo procesador. Los procesos ejecutados por los usuarios tendrán que compartir (por algún sistema de turnos) ese único procesador. A pesar de que estos sistemas están orientados a tareas **interactivas** (tareas donde un usuario humano ingresa las entradas esperadas por el software y espera por sus resultados), la rapidez de los procesadores (y la lentitud de los humanos) permite que cada usuario sienta que su tarea ha sido atendida de manera exclusiva por la máquina. Es decir, el usuario no solo no espera ser interrumpido, sino que no lo siente.

La aparición de máquinas con varios procesadores dio lugar a los sistemas **multiproceso** donde los procesos se distribuyen entre los procesadores disponibles. Sin embargo, note que esto no elimina la necesidad de la multiprogramación porque si el número de procesos a ejecutar supera el de procesadores disponibles, tendremos que recurrir al mismo sistema para compartir estos recursos. Por último, cuando un mismo proceso es capaz de utilizar varios procesadores, hablamos de un sistema de **paralelismo**.

Como la ejecución de un programa es un proceso, todo lo que se ejecuta en un computador es un proceso. Hay un proceso para Word, uno para la interfaz gráfica de Windows/MacOS, uno para cada juego al que le está dedicando más tiempo del que debería, etc. La ejecución de un proceso se debe ordenar entonces desde otro proceso. Necesitamos por lo tanto algún mecanismo para que un proceso pueda **crear** a otro. Adicionalmente, si tenemos varios procesos, seguramente vamos a querer **comunicarlos** (p.ej. Word necesita enviarle un documento al proceso que maneja la impresora para imprimirlo) y por último necesitamos una manera de **sincronizarlos** para asegurar que las cosas suceden en orden cuando sea necesario (no queremos que la impresora arranque a trabajar antes que le llegue el archivo que debe imprimir). La manera de pedirle cosas a la máquina es a través del sistema operativo, es el sistema operativo quien nos debe ofrecer maneras de invocar estos servicios. A la interfaz definida por el sistema operativo para pedir servicios se le conoce como *Application Programming Interface* (**API**).

Recordemos que un proceso es la ejecución secuencial de unas instrucciones que se encuentran en el código. Si queremos crear/comunicar/sincronizar procesos deben existir instrucciones especiales para invocar ese API del sistema operativo. Existen varias maneras de lograrlo: i) crear lenguajes que entre sus construcciones tengan algunas con ese objetivo. Tendremos que aprender ese nuevo lenguaje y contar por supuesto con un compilador para su traducción. Otra manera de lograrlo es, ii) con los lenguajes actuales, establecer un mecanismo para indicarle al compilador que cuando esté traduciendo, agregue estos llamados. El lenguaje no se altera, pero se introducen directivas, que el compilador entiende, para crear los respectivos llamados al API del sistema operativo. Por último, y es la manera como lo hace Java, iii) se puede disponer de una librería del lenguaje, que define un API del lenguaje y que es luego traducido por el compilador al API del sistema operativo. Cada una de estas aproximaciones tiene sus ventajas y desventajas. En el caso del curso utilizaremos la tercera aproximación porque así aprovechamos lo que usted ya ha aprendido en los cursos precedentes.

1.3 Concurrencia en Java

Java expone el API del sistema operativo a través de su propio API de lenguaje. El API de Java es un conjunto de clases que son parte del kit de desarrollo de Java (*Java Development Kit*, JDK) y que son escritas para proveer una gran cantidad de funcionalidad ya lista a los programadores. Entre esa funcionalidad se encuentra buena parte de los servicios del sistema operativo, pero no todos, por cuestiones de seguridad. Los que sí están por supuesto, son los que corresponden al manejo de la concurrencia, es decir los que permiten crear/comunicar/sincronizar procesos. En la terminología de Java a estos procesos se les conoce como *threads* (hilos de ejecución) así que seguiremos utilizando las dos nomenclaturas por ahora y ya veremos luego la diferencia.

Como Java es un lenguaje orientado a objetos, todo se representa por objetos y por lo tanto tenemos una clase `Thread` (si, la misma que utilizó sin entender al final de APO2). En esa clase existe un método `run()` que contiene el código a ejecutar por el thread. Cuando creamos un objeto de la clase `Thread` (operación de `new()`), el thread no se activa, solo se crea la estructura para representarlo. Para activarlo (que se empiece a ejecutar el código del método `run()`), debemos invocar el método `start()` sobre el thread. Se puede entonces crear una clase que extienda de `Thread` y que tenga toda esta funcionalidad o también se puede crear una clase que implemente la interfaz `Runnable` con un comportamiento similar. Solo se puede dar un único `start()` a un thread (no se puede enviar otro `start()` una vez termina). Si se quiere tener de nuevo otro thread, es necesario crear un nuevo objeto y ejecutar el `start()` en ese nuevo objeto.

*Nota 1: Un thread es un flujo de control sobre un código. El thread **no** es un objeto. Para tener un thread necesito un objeto (de la clase `Thread`) que lo represente. Una vez tengo ese objeto, puedo manipular el flujo de control.*

Los threads ejecutan su código y manipulan sus datos. Si cada uno tiene su propio código y sus propios datos no hay interferencia posible entre ellos. Si tiene el mismo código, pero trabajan sobre datos distintos, tampoco debería haber problemas porque al fin y al cabo el resultado de una ejecución es la alteración de los datos, no del código. Pero tenemos un problema si diferentes threads, con el mismo o con códigos distintos, trabajan sobre los mismos datos. Ahí sí podemos tener dificultades si los dos threads no se ponen de acuerdo para hacer las cosas.

Actividad 1-2: Suponga dos programas que manipulan una misma variable y se están ejecutando al mismo tiempo. El programa1 deja el valor de la variable en 5. El programa2 pone el valor de la variable en 6. Si el programa2 termina primero, ¿Qué valor podemos esperar en la variable cuando termine el programa1? ¿Habría manera de garantizar que el valor de la variable al final fuera el máximo de los dos valores?

Por supuesto dependiendo del código y el momento en que se ejecute este, las cosas pueden ser más complicadas. Recuerde que el código de un thread puede ser interrumpido en cualquier momento para dar paso a la ejecución de otro thread (eventualmente aquel con el que está compartiendo los datos). Suponga un thread que para realizar una acción debe evaluar que se cumplan unas ciertas condiciones. Si las condiciones están basadas en los valores de unas variables compartidas, podría pasar que el thread fuese interrumpido justo después de evaluar la condición, pero antes de realizar la acción que depende de dicha evaluación. Y como la ley de Murphy[1] siempre está presente, podría pasar que durante la interrupción se ejecute un thread que cambie

[1] La ley de Murphy es un enunciado basado en un principio empírico que trata de explicar, en términos algo pesimistas, que siempre debemos estar preparados para afrontar dificultades: "*Anything that can go wrong, will go wrong".*

los valores de las variables compartidas y que por lo tanto se deje de cumplir la condición que hacía que se tuviera que ejecutar la acción. Como cuando el primer thread recupere la posibilidad de ejecutarse lo hará en la instrucción siguiente a la última ejecutada, es decir no volverá a evaluar la condición; esto lo puede llevar, de manera equivocada, a ejecutar la acción.

El mecanismo de interrumpir procesos es iniciado directamente por el hardware, lo único que podemos garantizar es que cuando suceda, si nos encontramos en medio de la ejecución de una instrucción, esta instrucción terminará y luego sí el proceso en cuestión cederá el turno de ejecución. Las instrucciones a las que hacemos referencia entonces son las del procesador (¿recuerda assembler?), no las de alto nivel de un lenguaje como Java.

1.4 Sincronización

Nota 2: La sincronización hace referencia al conjunto de protocolos y mecanismos que son utilizados para preservar la integridad y consistencia de un sistema cuando varios procesos comparten datos en ese sistema.

La sincronización se ocupa del ajuste temporal de eventos para asegurar su ejecución en un orden predefinido. Al establecer el orden entre dos entidades en ejecución A/B, podemos identificar 4 situaciones diferentes: A y B se deben ejecutar al mismo tiempo, A y B no se pueden ejecutar al mismo tiempo, uno de los dos se debe ejecutar antes que el otro (A < B o B < A) o no hay restricción, es decir, el resultado de cualquier orden de ejecución es válido.

Las restricciones identificadas se dan por el acceso a los datos compartidos. Como ya dijimos, si no se comparten datos no hay necesidad de establecer ningún tipo de orden entre dos procesos. La parte relevante para establecer un orden en la ejecución de procesos es entonces el orden en el acceso a los datos compartidos. De las situaciones identificadas anteriormente, tres son claramente restrictivas y requerirán de protocolos y/o mecanismos específicos para asegurar su cumplimento.

- Que un proceso no se pueda ejecutar al mismo tiempo que otro hace referencia en realidad a que no accedan a datos compartidos al mismo tiempo. A esta restricción la llamaremos **exclusión mutua**.
- Que un proceso deba acceder a los datos antes que otro lo llamaremos **señalamiento**.
- Por último, cuando queremos que dos procesos accedan al mismo tiempo a los datos, lo llamaremos **encuentro**. Asegurar que dos procesos se ejecuten al tiempo se sale de las posibilidades del programador porque esa es una decisión que toma el sistema operativo en el momento de ejecutar los programas, basado en los procesadores disponible. El encuentro en realidad buscará que un proceso espere a que el otro tenga las condiciones necesarias para acceder a los datos y que por lo tanto el acceso efectivo ya no dependa sino de los recursos disponibles.

Actividad 1-3: Para cada una de las restricciones presentadas, identifique una situación de la vida real donde se manifieste.

1.4.1 Monitores

El primer mecanismo de sincronización que veremos es el de monitores que está orientado a asegurar la exclusión mutua. En exclusión mutua un conjunto de procesos que quiere manipular unos datos compartidos

debe sincronizarse para que, mientras un proceso accede a los datos, los otros no puedan hacerlo. La porción de código que ejecuta el acceso a los datos se conoce como **sección crítica**.

Los monitores, propuestos por Brich Hansen y Sir Charles Antony Richard Hoare, son construcciones de un lenguaje de programación que ofrecen sincronización a alto nivel, sin entrar en los detalles de la implementación necesaria para asegurarla.

Un monitor tiene un protocolo de entrada y de salida que asegura que, si alguien está dentro del monitor, cualquier otro proceso que intente entrar quedará en espera hasta que el primero salga y libere el monitor. Los procesos que quedan a la espera (i.e. bloqueados) lo hacen en una **cola de espera**. Si bien normalmente las implementaciones de estas colas de espera son justas (aseguran que alguna vez se obtendrá acceso), no siempre garantizan un orden FIFO (primero en llegar, primero en salir).

Todo el código asociado a un monitor está en exclusión mutua. Y a un mismo monitor se le pueden asociar varios métodos/procedimientos/funciones. Cuando un proceso está ejecutando uno de esos métodos, ningún otro proceso puede ejecutar ese, ni ninguno de los otros métodos asociados al mismo monitor. Al intentar ejecutar un método de un monitor que se encuentra ocupado (i.e. ya hay alguien ejecutando uno de sus métodos), el proceso queda detenido en la cola de espera del monitor. Cuando el monitor es liberado (i.e. quien estaba ejecutando el método del monitor termina de hacerlo), el proceso detenido podrá ingresar a ejecutar el método de interés y el monitor quedará de nuevo ocupado durante el tiempo que dura la ejecución de ese método. Si hay varios procesos detenidos en la cola se seleccionará solo uno de ellos para darle paso. El criterio de selección es dependiente de la implementación que haga el lenguaje.

La discusión anterior parece estar asociada a la existencia de múltiples procesadores. ¿De qué otra manera podría entenderse la preocupación por no ejecutar dos procesos al mismo tiempo? En realidad, la sincronización también es necesaria en el caso de las máquinas monoprocesador porque, como ya habíamos dicho, si se mira con un lente grueso (más de un segundo), en un único procesador puede haber varios procesos corriendo al mismo tiempo (multiprogramación). En este caso el monitor asegura que, si uno de los procesos debe ceder el procesador a otro, estando en una sección crítica, cualquier proceso que intente ingresar a esa sección crítica quedará bloqueado en la cola de espera. Una vez el proceso que se encontraba en la sección crítica vuelva a obtener acceso al procesador y termine la ejecución del método sincronizado, la cola de espera podrá liberar a uno de los procesos allí bloqueados.

En Java los monitores están implementados en todas las clases. Cada objeto que se crea en una clase es un monitor. Y la exclusión mutua se asegura automáticamente entre los métodos que tengan el atributo `synchronized`. Es importante entender que es el objeto y no la clase el que implementa el monitor (bueno, en realidad la clase también, pero eso lo veremos luego). Lo anterior implica que solo se garantiza exclusión mutua entre métodos `synchronized` del mismo objeto. Dos objetos distintos de la misma clase son monitores diferentes y por lo tanto no hay sincronización entre ellos.

Actividad 1-4: Ejecute varias veces el siguiente código y explique el resultado. Asegúrese de entender bien la sintaxis y la estructura. ¿Es necesario el `static` en la declaración de `oMax`? ¿Cuál es el propósito de `synchronized` en el método anotar? Pruébelo con y sin ese synchronized.

```java
public class T extends Thread {

    private static Maximo oMax = new Maximo () ;
    private int num = 0 ;

    public T (int n) {
        num = n ;
    }

    public void run () {
        oMax.anotar(num) ;
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++)
            new T (i).start();

        System.out.println ("El máximo es: " +
                            oMax.darMaximo()) ;
    }
}
```

```java
public class Maximo {

    private int maximo = 0 ;

    public int darMaximo () {
        return maximo ;
    }

    public synchronized void anotar (int n) {
        if (n > maximo)
            maximo = n ;
    }
}
```

Actividad 1-5: Corrija el siguiente código para que la salida sea los números del 1 al 10 ordenados sin repetidos y sin saltos.

```java
public class T extends Thread {

    public void run () {
        Lista.sumar() ;
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++)
            new T ().start() ;


    }
}
```

```java
public class Lista {

    private static int suma = 0 ;

    public static void sumar () {
        suma ++ ;
        System.out.println ("Suma: " + suma) ;
    }
}
```

1.4.1.1 Secciones críticas

Los monitores protegen de la ejecución simultánea de métodos, convirtiendo a estos métodos en secciones críticas o regiones críticas de un programa. Pero no siempre queremos que todo el código de un método sea una sección crítica, a veces queremos una granularidad más fina y poder demarcar de manera más flexible el inicio y fin de una sección crítica. Incluso podríamos querer que en un mismo método haya partes del código que estén sincronizadas y partes que no.

Para asegurar la sincronización entre threads de manera flexible, Java ofrece los **bloques de código sincronizados**. Los bloques de código permiten que se definan secciones críticas que incluyan desde una línea de código hasta tantas líneas como se desee. Para definir un bloque de código, todo lo que se debe hacer es usar la construcción `synchronized (objeto)` para marcar el bloque. Note que la construcción `synchronized` recibe como parámetro un objeto. Este objeto actúa como monitor para la sincronización del bloque de código, es decir, el código está sincronizado en ese objeto. Esto es equivalente a lo que habíamos ya visto sobre los monitores definidos al sincronizar métodos enteros, la sincronización se hace siempre asociada a un objeto.

Actividad 1-6: ¿Qué diferencia habría entre la ejecución de estos dos programas? Note que siempre necesitamos objetos para poder definir secciones críticas.

```
public class Accion {

    private int x = 0 ;
    private int y = 1 ;
    public void accionar () {
        synchronized (this) {
            this.accion (x) ;
            this.accion (y) ;
        }
    }
}
```

```
public class Accion {
    private int x = 0 ;
    private int y = 1 ;

    private Integer objX = new Integer(0) ;
    private Integer objY = new Integer(1) ;

        public void accionar () {
            synchronized (this.objX) {
                this.accion (x) ;
            }
            Synchronized (this.objY) {
                this.accion (y) ;
            }
        }
}
```

1.4.2 Eventos

Las secciones críticas nos permiten resolver la restricción de exclusión mutua, una de las restricciones que deben asegurarse cuando tenemos concurrencia. La exclusión mutua nos permite garantizar que no haya dos procesos o más manipulando el mismo conjunto de datos al mismo tiempo. Pero esta garantía no impone ningún orden en el acceso, solo sabemos que los dos procesos no estarán accediendo al mismo tiempo. Si queremos asegurar que un proceso en particular manipule primero un conjunto de datos y solo después lo pueda hacer otro, lo que habíamos llamado señalamiento, necesitamos un mecanismo distinto: el manejo de **eventos**.

Los eventos pueden ser implementados directamente por el programador, utilizando una sincronización por variables compartidas. Revise el siguiente código:

```
private MySignal sharedSignal = ...

   while (!sharedSignal.darHayDatos () {
      // No haga nada…espera activa
}
```

```
public class MySignal {

    private boolean hayDatos = false ;

    public synchronized boolean darHayDatos () {
        return hayDatos ;
    }

    public synchronized void modHayDatos (boolean data) {
        hayDatos = data ;
    }
}
```

Recuerde que lo sistemas modernos son de multiproceso, es decir, alternan los procesadores entre los distintos procesos en ejecución. En este caso, lo que tenemos son dos procesos que se están sincronizando a través de un objeto de la clase `MySignal`. En particular, el proceso que ejecute el código de la izquierda no avanzará (seguirá evaluando la condición del `while` todo el tiempo que le corresponda el procesador) hasta tanto otro proceso no modifique el valor del atributo `hayDatos`. Esta solución cumple con el requerimiento de señalamiento porque si lo que queremos es que un proceso se ejecute estrictamente antes que otro todo lo que hay que hacer es incluir el `while` de la izquierda en el código de aquél que se espera se ejecute después. Si un segundo proceso no ejecuta el código de la izquierda, será libre de iniciar su ejecución cuando el procesador está disponible y todo

lo que tiene que hacer es invocar en algún momento el método `darHayDatos(true)` para darle paso al que está esperando.

El problema con la solución anterior es que desperdicia el tiempo del recurso más valioso de un computador: el procesador. Note que un proceso que debe esperar lo hace consumiendo procesador. El proceso debe utilizar todo su tiempo de procesador haciendo la misma pregunta y, por supuesto, obteniendo la misma respuesta puesto que el método `dayHayDatos()` está sincronizado con `modHayDatos()`, así que ni siquiera en la presencia de un segundo procesador se podría modificar el valor del atributo `hayDatos`. A una espera de este tipo se le conoce como **espera activa**. Pero peor aún, si cuando se le termine su tiempo de procesador (algo que ocurre sin nuestro control y por lo tanto en cualquier instante) el proceso se encuentra dentro del método `darHayDatos()`, liberará el procesador, pero sin liberar la sección crítica (siempre tenga presente la ley de Murphy). Esto haría de nuevo imposible modificar la variable `hayDatos`.

Los eventos son mecanismos de sincronización que se utilizan para que un proceso pueda indicar que espera que se cumpla una cierta condición o para indicar que ya se ha producido la condición esperada. Para dejar en espera a un proceso que necesita el cumplimento de una condición particular para poder seguir adelante se utiliza una cola de espera similar a la de los monitores. Los eventos se implementan entonces como estructuras con un estado, una cola y con operaciones para manipular o consultar ese estado. El manejo de la cola es interno de los eventos por lo cual no necesitamos operaciones para manipularla. Un proceso que está en la cola de espera no consume procesador, de alguna manera es como si quedara inactivo mientras ocurre el evento que espera.

Cuando se combinan los eventos con los monitores es importante entender la cooperación que se da entre los dos. Si dentro de un bloque de código sincronizado se llama a una de las operaciones de los eventos, tenemos una situación donde un proceso, habiéndose apropiado de un monitor, puede quedar a la espera de que se cumpla una cierta condición. Esto podría llevar a que el proceso pasara a la cola de espera del evento sin liberar el monitor para que otro proceso lo pueda usar. Afortunadamente, esta situación está prevista por las implementaciones de los lenguajes y cuando un proceso pasa a la cola de espera de un evento, estando dentro de un monitor, libera primero el monitor y luego si se queda a la espera del evento particular. Esto por supuesto obliga a un manejo muy cuidadoso por parte del programador para no afectar las condiciones esperadas de ejecución de los procesos.

Los eventos en Java se conocen como señalamiento porque eso es lo que exactamente se maneja en el lenguaje: señales. Un thread en Java puede enviar una señal a otro o puede esperar a que le llegue una señal de algún thread. Esta espera se dice **pasiva** porque el thread que debe esperar lo hace sin consumir procesador. Java tiene un mecanismo interno para desactivar a un thread que debe esperar. Una vez algún otro thread genera la señal esperada, Java vuelve a activar el thread en espera para que pueda continuar con su ejecución.

Los métodos ofrecidos por Java, y definidos en la clase `java.lang.Object`, son `wait()`, `notify()` y `notifyAll()`. Estos métodos están disponibles para todos los objetos y como la sincronización está asociada a los objetos, cuando un thread llama el método `wait()` de un objeto, quedará inactivo hasta tanto otro thread llame al método `notify()` (o `notifyAll())` de ese mismo objeto. Para poder llamar a `wait()` o a `notify()` un thread debe primero apropiarse del objeto, esto es, estar dentro de un bloque sincronizado por ese objeto. Cuando un thread llama a `notify()` sobre un objeto, uno de los threads esperando sobre ese objeto será activado. Por su parte, `notifyAll()` activará todos los threads esperando sobre ese objeto. Como `wait()` puede lanzar excepciones (en particular si se invoca sin haber apropiado el objeto sobre el que se va a sincronizar), este llamado siempre debe ir en un bloque `try/catch`.

Una vez un thread es activado, este debe volver a apropiarse del monitor asociado al objeto que lo invocó. Para ello debe esperar que el thread que generó la señal abandone el monitor (bloque de sincronización) y así poder pasar a la cola del monitor para competir por acceso en caso de que haya más threads allí esperando. Cuando se llama a `notifyAll()`, todos los threads esperando por esa señal serán pasados a la cola del monitor y competirán por acceso entre ellos y con aquellos que estuvieran ya en la cola. Si hay varios threads esperando y llega una señal (`notify`()), Java nuevamente no garantiza el orden y selecciona a uno cualquiera de los threads para ser activado y pasar a la cola de espera del monitor.

Java no guarda las señales enviadas en caso de que no haya ningún thread esperándolas. Estas señales se pierden, lo que implica que si el `notify()` se ejecuta antes que el `wait()`, el proceso que espera puede hacerlo por siempre si nadie más reenvía esa señal. Por supuesto, se podrían construir métodos nuevos que a partir de los de Java, cambien este comportamiento.

A continuación, presentamos una posible implementación de sincronización entre dos procesos utilizando los mecanismos disponibles en Java.

```
public class ObjetoMonitor {
}

public class MyWaitNotify {
    private ObjetoMonitor objMon = new ObjetoMonitor () ;

    public void esperar () {
        synchronized (objMon) {
            try {
                objMon.wait () ; // espera pasiva
            } catch (InterruptedException e) { … }
        }
    }
}
```

```
    public void avisar () {
        synchronized (objMon) {
            objMon.notify () ;
        }
    }
```

Similar a lo que habíamos visto antes, para sincronizarse dos procesos entre ellos, el proceso que debe esperar llama al método `esperar ()` y el que debe ejecutarse antes no llama a este método. En lugar de eso, ejecuta lo que tenga que hacer y cuando esté listo para dar paso al primero, todo lo que debe hacer es invocar el método `avisar ()`. La espera pasiva es más eficiente porque al no consumir procesador inútilmente, éste queda disponible para que otros procesos (y en particular aquél que debe avisar) accedan más rápidamente a este recurso.

Actividad 1-7: Escriba un método `miNotify()` y otro `miWait()` que permitan que si una señal se lanza y nadie está esperando por ella, no se olvide. ***Ayuda****: cree una clase nueva con un atributo booleano.*

1.4.3 Barreras

Una vez presentadas las soluciones de alto nivel para restricciones de exclusión mutua y señalamiento, nos queda por revisar una solución para los encuentros. Una barrera es el mecanismo que utiliza un grupo de procesos para definir un punto en el código fuente en el cual todos los procesos deben detenerse; no pueden avanzar hasta tanto todos los otros procesos alcancen este mismo punto, es decir esta barrera.

La implementación básica de una barrera es un estado (abierta/cerrada) y un contador para saber el número de procesos que han llegado a la barrera (y que se encuentran esperando a que llegue el último del grupo). La barrera se inicializa en estado cerrada y el contador en el número de procesos del grupo. Cada vez que llega un nuevo proceso a la barrera, el contador disminuye. Cuando el contador llega a cero (0), todos los procesos son reactivados y podrán continuar con la ejecución del código en función de los procesadores disponibles. No hay ninguna garantía sobre el orden en que se ejecutarán después de la barrera.

Repasemos la *Actividad 4*. El problema se presentaba porque el thread principal podía escribir el resultado antes que todos los 10 threads hayan completado su tarea de actualización del máximo. Incluso, podría pasar que algún thread no hubiese empezado a ejecutarse (a pesar de estar creado y arrancado). Una solución podría ser una barrera.

Java ofrece la clase `java.util.concurrent.CyclicBarrier` para manejar barreras. Los objetos de esta clase no solo tienen el contador de threads sino que opcionalmente pueden recibir una acción para ser ejecutada una vez todos los procesos lleguen a la barrera. El método `await()` hace que un thread espere, si cuando lo ejecute, el contador de la barrera aún no ha llegado a cero (0).

Java también ofrece el método `join()` que espera por la terminación de un thread. Este método hace que el thread en ejecución se detenga hasta que el thread al que quiere esperar termine. Al igual que `await()`, `join()` acepta que se le especifique un tiempo máximo de detención en la barrera.

A continuación, un ejemplo de uso de la barrera. En este ejemplo, se crean N threads y los N esperan a que todos estén creados y listos para ejecutar para poder iniciar.

```
public void run () {
   try {
       barrera.await();  // Se queda esperando hasta que N threads hagan esta llamada.
   } catch (Exception e) {
       ...
   }
    // Aquí el código del thread.
}
```

```
// Creación de la barrera para N threads
CyclicBarrier barrera = new CyclicBarrier(N);
```

Actividad 1-8: Modifique el código anterior para que el inicio de la ejecución de estos threads esté controlada por el thread que los crea. Solo deben iniciar su ejecución, una vez el thread principal lo autorice. Escriba el código completo (i.e. main y declaración de clases) para asegurarse que entiende todos los elementos de la solución.

1.5 Implementación de la concurrencia

Los monitores, los eventos e incluso las barreras son construcciones de alto nivel que los lenguajes de programación exponen para facilitarle al programador resolver los problemas de sincronización. Y estas construcciones se apoyan en servicios del sistema operativo para poder ser implementadas de manera eficiente. No es posible para un lenguaje de programación ofrecer servicios de esperar, avisar, bloquear, activar ni ninguno de estos verbos que hemos utilizado informalmente a lo largo de este documento. El único que puede garantizar

estos comportamientos es el sistema operativo y por eso debemos entonces formalizar mejor estos términos y entender qué es lo que está pasando detrás de escenas.

1.5.1 Estados de un proceso

La multiprogramación abrió la posibilidad de compartir un procesador entre varios procesos y la manera de lograrlo es que el sistema operativo asegure esta gestión. El sistema operativo debe tener control de todos los procesos en el sistema y asegurar que el procesador se use de la manera más eficiente posible. Algunos principios básicos que debe respetar un sistema operativo son:

- Si un proceso no tiene todo lo que necesita para su ejecución (por ejemplo, si espera un dato del disco o una señal de otro proceso), no debe acceder al procesador.
- El procesador no debe estar ocioso a menos que no haya ningún proceso con todo lo necesario para su ejecución.
- Un proceso no debe apropiarse indefinidamente del procesador. Esto para permitir que otros tengan opción de ejecutarse.
- Un proceso que tenga todo lo que necesita para su ejecución debe acceder *pronto* al procesador. Qué tan *pronto* dependerá del nivel de interactividad y la carga del sistema.

Respetar estos principios implica que el sistema operativo debe conocer el estado de todos los procesos. Por cada proceso debe saber: si está usando o no al procesador, cuánto tiempo lleva en el procesador, si tiene todo lo que necesita para su ejecución, si está esperando algo y en ese caso, qué está esperando, cuándo llega algo de lo esperado, etc.

Un sistema operativo maneja 3 estados principales: Ejecutando (E), listo (L) y dormido (D), tal y como se presenta en la *Figura 1-1 Estados y transiciones de un proceso*. El estado E corresponde a un proceso que está utilizando el procesador; en una máquina monoprocesador solo podrá haber un único proceso en este estado en un instante dado. La lista de procesos a los cuales solo les falta que el procesador esté disponible para poderse ejecutar se encontrará en el estado L. Hablamos de una lista porque hay una noción de orden en el conjunto de procesos en estado L. Por último, el estado D corresponde a los procesos a quienes no les serviría que les asignen procesador porque de todas maneras les hace falta algo adicional (i.e. que suceda un evento). Estos procesos dependen de algo externo a ellos para poder continuar su ejecución.

El sistema operativo controla las transiciones entre estos estados. Al momento de crear un proceso lo deja en el estado Listo, cuando el procesador está disponible ejecuta un algoritmo para seleccionar el proceso que debe pasar de Listo a Ejecutando. En Ejecutando pueden suceder tres eventos que obliguen una transición: si el sistema operativo determina que el proceso en Ejecutando lleva demasiado tiempo y que es hora de compartir el procesador con otro proceso (ya ha pasado un tiempo que se conoce como “quantum”), moverá al proceso de Ejecutando a Listo; de otra parte, podría ocurrir que el proceso en Ejecutando solicite esperar por algún evento (o necesite un resultado que se pueda demorar mucho, como una operación de entrada/salida), en este caso el proceso pasa al estado Dormido; por último, podría simplemente pasar que el proceso en Ejecutando termine (ejecute la última instrucción); en este caso el proceso debe salir del sistema. Cuando el sistema operativo detecta que ha sucedido un evento por el cual está esperando un proceso en el estado Dormido, mueve este proceso de nuevo al estado Listo. Allí competirá con los otros procesos que tienen todo para ejecutarse y así acceder al procesador.

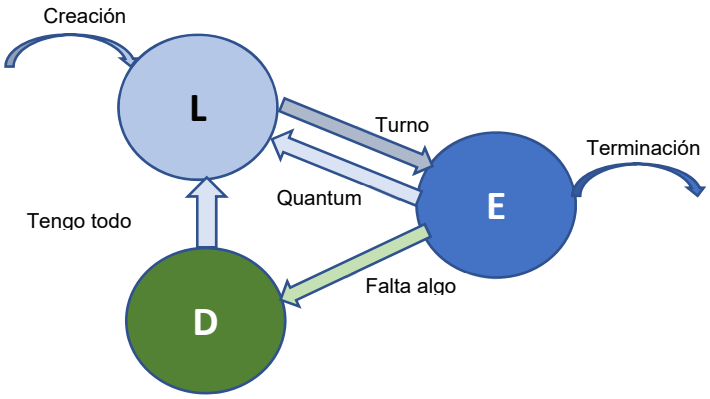

Figura 1-1 Estados y transiciones de un proceso

Actividad 1-9: Describa con sus propias palabras por qué no son posibles las transiciones entre D y E ni entre L y D.

La sincronización se logra entonces por las transiciones entre estos estados. Cuando un proceso quedaba en una cola de espera, nos referíamos en realidad a que quedaba en el estado Dormido. Cuando sucedía el evento esperado por un proceso, lo que sucedía es que el proceso pasaba al estado Listo y allí entraba a competir con los otros en ese estado para acceder al procesador.

1.5.2 Semáforos

Los problemas de sincronización están presentes desde que aparece la multiprogramación y esto quiere decir que son anteriores a los lenguajes estructurados de alto nivel (principios de los años setenta). Las soluciones de monitores, eventos y barreras son mecanismos de alto nivel desarrollados para estos lenguajes estructurados pero que necesitan el soporte del sistema operativo para su implementación.

Los **semáforos** fueron propuestos por Edsger Dijkstra a finales de los sesenta como un mecanismo único para resolver todos los problemas de la sincronización, lo que significa que permiten imponer restricciones de exclusión mutua, señalamiento y encuentro a los programas. Los semáforos son el mecanismo implementado por los sistemas operativos para soportar las soluciones de más alto nivel ofrecidas por los lenguajes de programación. Para ello, los sistemas operativos ofrecen dos rutinas que hacen parte del estándar POSIX (norma de la IEEE que define una interfaz estándar para los diferentes sistemas operativos): `sem_wait()` y `sem_post()` que corresponden a los originales P() y V() definidos por Dijkstra[2]. Un semáforo está compuesto por un entero, para representar el número de recursos disponibles, y una cola para mantener una referencia a los procesos que esperan la liberación de los recursos controlados por el semáforo.

Para que un semáforo pueda coordinar el acceso a un conjunto de recursos, todos los procesos que se desean sincronizar deben compartir el mismo semáforo y el semáforo debe inicializarse con el entero en el número de recursos disponibles inicialmente y con la cola vacía. Para acceder a un recurso, un proceso debe invocar la rutina P() sobre el semáforo compartido y si no existe disponibilidad del recurso será enviado al estado Dormido. Por su parte, cuando un proceso libera uno de los recursos, invocará la rutina V() del semáforo que se encargará de

[2] P por “passering” y V por “vrijgave”, que a uno no le dicen nada a menos que hable holandés.

la transición al estado Listo de uno de los procesos esperando por ese recurso. A continuación, presentamos el pseudo-código de P() y V().

```
P (s) {
    s.contador--
    if (s.contador < 0) {
       s.cola ← proceso
       dormir proceso
    }
}
```

```
V (s) {
    s.contador++
    if (s.contador <= 0) {
       q ← s.cola
       despertar q
    }
}
```

Las operaciones dormir y despertar referenciadas en el pesudo-código son operaciones que solo puede implementar el sistema operativo porque incluyen la manipulación del estado del proceso. Pero más delicado aún, para que estas operaciones P() y V() funcionen bien, cada una de ellas debe ejecutarse de manera atómica, esto es, necesitamos garantizar que una vez se comienzan a ejecutar y manipular la estructura del semáforo, las rutinas terminan sin que otro proceso pueda invocar acciones sobre esa estructura. Lo anterior es parecido al requerimiento de exclusión mutua, pero más fuerte. Si P() y V() se implementaran a alto nivel, deberían estar en un monitor porque están manipulando variables compartidas (el contador y la cola), pero no podemos recurrir a los monitores porque estamos a nivel de sistema operativo y justamente estamos definiendo los mecanismos básicos para implementarlos. El sistema operativo tiene una capacidad, que veremos en detalle más adelante, que le permite asegurar que una vez empiezan a ejecutarse estas rutinas, el proceso que las está ejecutando no sufre ninguna transición de Estado (permanece en Ejecutando). Si, por ejemplo, al proceso en ejecución se le acabara el tiempo de procesador en medio de la ejecución de estas rutinas, el sistema operativo garantiza que el proceso no cede el procesador hasta tanto no se terminen de ejecutar todas las acciones de la sección crítica.

Java, además de lo que ya vimos, ofrece también un API para usar este mecanismo de semáforos. La clase `java.util.concurrent.Semaphore` ofrece los métodos `acquire()` y `release()` que son las versiones de Java para P() y V(). Ambos métodos aceptan un parámetro entero si se desea especificar un número de recursos a reservar o liberar de manera atómica. Existe también el `tryAcquire()` para intentar reservar sin bloquearse si no hay disponibles recursos.

El API de POSIX define seis operaciones que deben ser soportadas por los sistemas operativos:

- `sem_t s`: declaración de un semáforo
- `int sem_init (sem_t *s, int shared, unsigned int n)`: inicialización
- `int sem_wait (sem_t *s)`: equivale a P
- `int sem_post (sem_t *s)`: equivale a V
- `int sem_getvalue (sem_t *s, int *valn)`: retorna el valor del contador
- `int sem_destroy (sem_t *s)`: destruye el semáforo

Según vimos, Java implementa la sincronización con mecanismos de alto nivel. Para ello, todo objeto que se crea en Java tiene automáticamente asociado un semáforo (de exclusión mutua, inicializado en 1). Este semáforo controla el acceso a los métodos sincronizados. Un semáforo de exclusión mutua asegura que solo un proceso puede ejecutar el código protegido por el semáforo, esto se lograr inicializando el contador del semáforo en 1. Note que un método sincronizado no es más que un método en donde se hace P() del semáforo del objeto al principio del método y V() al final. Adicionalmente, todo objeto tiene asociado un semáforo para eventos (que

se controla con los métodos `wait()` y `notify()`). Por último, cada clase también tiene un semáforo de exclusión mutua para implementar monitores con métodos estáticos.

Implementar la sincronización por eventos se puede hacer directamente con semáforos también. Un `wait()/notify()` se puede implementar con un P()/V() sobre un semáforo asociado. Para una exitosa implementación, habría que tener en la lógica del programa un contador que permitiera saber el número de procesos en estado Dormido esperando por el evento en particular y asegurarse que antes de hacer un P() en el semáforo del evento, se realiza un V() sobre el semáforo de exclusión mutua para liberar el monitor donde generalmente se encuentra.

1.5.3 La concurrencia en Java

Java es muy rico en opciones para manejo de la multiprogramación. `java.util.concurrent` es un paquete agregado en la versión 5 que trae múltiples construcciones para ayudar a desarrollar aplicaciones concurrentes. Antes de esta versión, el programador debía implementar sus propias clases para resolver todos los problemas. Presentamos aquí algunas clases que pueden ser de utilidad para el lector y lo invitamos a revisar la documentación de Java para su utilización. Una buena referencia para el manejo de la sincronización en Java es el tutorial de Jacob Jenkov disponible en http://tutorials.jenkov.com/java-util-concurrent/index.html

1.5.3.1 Las clases atómicas

Las clases atómicas (`atomic`) son un paquete de clases que ofrecen objetos que pueden ser leídos y/o modificados de manera atómica. Recuerde que esto significa que la variable puede ser manipulada sabiendo que el proceso que lo está haciendo terminará la manipulación sin que otro proceso pueda interferir en esa manipulación. Disponer de métodos que tienen esta capacidad elimina muchas de las necesidades de sincronización adicional. Además de consultar y asignarle un valor al atributo de estos objetos, Java ofrece una operación de `getAndSet()` que, como su nombre lo indica, obtiene el valor actual de la variable y la modifica con un nuevo valor, todo en una operación atómica.

Existen objetos atómicos con atributos booleanos, enteros, long, referencias e incluso arrays de estos mismos tipos. La clase `AtomicInteger` y la clase `AtomicLong` disponen de métodos para sumar y restar valores al atributo de la clase, operaciones que por supuesto, son atómicas.

Actividad 1-10: Revise la documentación de Java sobre la sintaxis y semántica de los métodos de la clase `AtomicInteger`

1.5.3.2 Los objetos de tipo Cola

Java también ofrece distintas clases de manipulación de colas con diferentes patrones. En una cola bloqueante de Java, las operaciones de inserción y eliminación de elementos pueden terminar generando la transición al estado Dormido del proceso que las invoque; por ejemplo, si un proceso intenta retirar de una cola vacía o agregar un elemento a una cola llena (y se ha usado una clase que restringe el tamaño de la cola). Esta cola implementa entonces el patrón productor-consumidor que ya vimos en clase. Algunas clases permiten definir un tiempo máximo de bloqueo (`DelayQueue`), otras aseguran orden de inserción y eliminación (`LinkedBlockingQueue`), la clase `PriorityBlockingQueue` permite definir una cola de prioridades, la

clase `SynchronousQueue` sincroniza con un encuentro el acceso a una cola de un solo elemento y en la clase `BlockingDeque` se implementa una cola en la que se pueden agregar y eliminar elementos por ambos extremos de la cola. La mayoría de las implementaciones ofrece métodos que permiten modificar el comportamiento por defecto aquí descrito.

1.5.3.3 Los candados

La clase `Lock` permite implementar los bloques de códigos sincronizados, pero es más flexible porque habilita la posibilidad de que un thread diferente al que inició el bloqueo sea el que lo termine. Ofrece los métodos `lock()` y `unlock()`. El primero se usa para definir el inicio de una sección crítica y el segundo define el fin de esa sección crítica. Cada objeto de la clase `Lock` puede definir una sección crítica, pero con la ventaja de no tener que hacer los dos llamados en el mismo método. La clase `ReadWiteLock` implementa el patrón de lectores/redactores.

1.5.3.4 Otras estructuras

Por último, Java tiene las clases `ConcurrentHashMap` y `CopyOnWriteArrayList` que implementan estructuras de hash y arrays con diferentes niveles de concurrencia (algunos incluso definidos por el programador). Animamos al lector a tener presente todas estas posibilidades a la hora de resolver problemas de concurrencia en Java.

1.6 Retos de la concurrencia

Como seguramente ya se habrá dado cuenta, el manejo de programas que comparten datos es muy delicado y propenso a errores. Por esta razón, muchos programadores prefieren desarrollar sus programas con procesos aislados que no interactúan entre ellos y así no tener que lidiar con ninguno de los mecanismos que hemos visto hasta ahora. El resultado de eso son aplicaciones que no aprovechan las ventajas de la concurrencia y que ni siquiera son capaces de explotar adecuadamente los potentes procesadores presentes hoy en día en los dispositivos personales (que como mínimo tienen 2 cores).

Para muchas aplicaciones en cambio, el uso de la concurrencia no es opcional y lo mejor es tener bien identificadas las dificultades a los que nos enfrentamos. Esta sección cierra el tema de sincronización revisando los diferentes problemas que nos podemos encontrar cuando utilizamos los distintos mecanismos de sincronización ofrecidos por los lenguajes y el sistema operativo.

Lo primero que debemos recordar es que no todos los programas que comparten datos necesitan sincronización. Existen programas en donde el orden de acceso a los datos por parte de los diferentes procesos que cooperan para obtener el resultado no es relevante para determinar su corrección. Esto no significa que cualquier orden de ejecución conduzca siempre al mismo resultado (que podría ser), sino que todos los resultados posibles son válidos y garantizan la corrección del programa. Un programa que presenta esta característica se conoce como algoritmo o modelo no determinista. Buscar un valor en una matriz y reportar la fila en la que se encuentra sería un ejemplo de este modelo. SI el valor se encuentra en diferentes filas y lanzamos un proceso distinto a buscar en cada fila, la ejecución repetida de este programa podría arrojar resultados diferentes, todos correctos. En general, los algoritmos que involucran probabilidades y/o aleatoriedad son no deterministas.

De otra parte, encontramos algoritmos donde la corrección del resultado depende del orden en que se ejecuten los procesos cooperativos. Decimos entonces que los procesos tienen **condiciones de carrera** que requieren ser resueltas por medio de la sincronización. Todos los ejemplos que utilizamos para explicar los mecanismos de sincronización experimentan condiciones de carrera.

Una de las soluciones que vimos como posible solución a los problemas de sincronización es la espera activa. Esta técnica verifica repetidamente una condición, consumiendo procesador y evitando que otros procesos puedan usar el procesador (incluso aquél que puede hacer que se cumpla la condición). El programa que presentamos en la sección 1.4.2 ilustraba la espera activa y lo usamos para introducir los beneficios de resolver ese tipo de situaciones por sincronización por eventos. Lo que no mencionamos en ese momento es que recurrir a rutinas de sincronización también tiene su costo y que, en algunos casos, puede ser conveniente realizar una espera sin pasar al estado Dormido. Cuando un proceso hace un `wait()`, el resultado es que pasa al estado Dormido y luego cuando sucede el evento pasará al estado Listo donde deberá competir como un recién llegado con los procesos que ya se encuentran ese ese estado. Esto puede terminar en una espera excesiva ya que, aunque el evento por el que espera un proceso suceda rápidamente, el tiempo total para acceder al procesador puede ser muy alto. Recuerde que los cambios de estado son mediados por el sistema operativo y seguro no son gratis.

Un caso donde la espera activa puede ser adecuada es el de las máquinas multiprocesadores, cuando de alguna manera sabemos que el proceso que genera el evento ya se está ejecutando en otro de los procesadores de la máquina. En ese caso es muy posible que el evento a esperar esté a punto de suceder y no valga la pena ceder el procesador porque ya casi se va a cumplir la condición que está esperando el proceso. Esperar a que esto suceda manteniendo el control del procesador puede ser una estrategia válida para acelerar la ejecución del programa completo.

Java ofrece la posibilidad de tener una solución intermedia entre la espera activa (quedarse en el estado Ejecutando) y la espera pasiva (pasar al estado Dormido). El método `yield()` en la clase `Thread` se utiliza para ceder el procesador sin pasar por el estado Dormido. Un thread que ejecuta el método `yield()`, liberará el procesador y regresará directamente al estado de Listo. Esto es equivalente a no consumir todo el quantum al que tiene derecho un thread porque necesita algo y sabe (o sospecha) que el thread que lo puede ayudar está listo para ejecutar. Este método lo utilizan los desarrolladores de aplicaciones concurrentes para facilitar que sus threads avancen a un ritmo más controlado por el programador. Otro método de la misma clase es `sleep()` que también se usa para ceder el procesador, pero esta vez esperando en el estado Dormido por un tiempo determinado por el parámetro del método.

Otro de los problemas que vimos en las soluciones presentadas es la inanición. Esta situación se presenta cuando la implementación de la sincronización permite que, en situaciones particulares, uno o varios procesos queden sin posibilidad de ejecutarse por la continua aparición de otros procesos a los que se les da algún tipo de prioridad. En el ejemplo de lectores-redactores, una vez los lectores estaban accediendo al archivo, los redactores debían esperar a que todos los lectores terminaran, pero como siempre podían aparecer nuevos lectores, esa espera podía ser infinita.

Lograr la sincronización deseada en una aplicación concurrente no es evidente. Que se garantice la exclusión mutua, el señalamiento adecuado o los encuentros precisos, haciendo un buen uso de los recursos y sin generar problemas de inanición, requiere de mucho cuidado al momento de programar y de pruebas extensivas a la hora de validar la funcionalidad. Aparte de estas dificultades, otro problema que sucede en muchos sistemas

concurrentes con protocolos complejos de sincronización son los interbloqueos (*deadlocks*). Un interbloqueo sucede cuando, por ejemplo, un proceso P1 espera por un evento que debe generar un proceso P2, pero a la vez P2 espera por un evento o recurso que controla P1. Una dificultad de los interbloqueos es que no siempre suceden, sino que justamente aparecen en condiciones muy particulares en el orden de ejecución de los procesos.

Existen dos tipos de interbloqueos: bloqueos activos o vivos en los que los procesos no avanzan porque intentan continuamente ejecutar una acción que fracasa (p.ej. apropiarse de un recurso). Se dice que son vivos porque los procesos hacen continuamente el ciclo Ejecutando-Dormido-Listo-Ejecutando. Los interbloqueos pasivos, por su parte, son aquellos en los que los procesos quedan atrapados en el estado Dormido y nunca logran salir de él. Pueden suceder cuando un proceso se apropia de un recurso y nunca lo libera de nuevo, o simplemente porque nunca sucede el evento por el que un proceso se quedó esperando.

El problema de los interbloqueos ha sido muy estudiado en la literatura, en particular, porque se manifiesta frecuentemente en las aplicaciones complejas y un ejemplo de dichas aplicaciones es el sistema operativo mismo. En estos grandes sistemas, los interbloqueos suceden porque se generan dependencias circulares entre varios procesos y eso es muy difícil de prevenir en la codificación de un programa multiproceso, en particular si, como en el caso de los sistemas operativos y los drivers, este código no es escrito por el mismo conjunto de desarrolladores. Se han identificado cuatro condiciones necesarias para que ocurran los interbloqueos:

- Los recursos se pueden usar en Exclusión Mutua
- Los procesos solicitan recursos mientras tienen otros (*hold and wait*)
- Hay apropiación (en realidad, el problema es que no haya “expropiación”)
- Hay un ciclo de dependencias

Si alguna de las condiciones anteriores no se cumple, no hay interbloqueos. Desafortunadamente, las tres primeras son necesarias para controlar la sincronización de procesos concurrentes, la única que podríamos atacar es la generación de cadenas de procesos en la cual un proceso se apropia de uno o más recursos que son solicitados por un proceso posterior en la cadena. La formación de una cadena circular puede conducir a un interbloqueo.

Son los sistemas operativos los que ofrecen los servicios de sincronización, por lo tanto, a ellos les corresponde lidiar con la aparición de estos interbloqueos. Un interbloqueo a nivel del sistema operativo puede causar el congelamiento de la máquina. Los sistemas operativos pueden seguir una de las siguientes estrategias para manejar los ciclos de dependencias que dan lugar al origen de los interbloqueos:

- Prevenir: no ofrecer servicios que puedan generar ciclos
- Evitar: detectar que una acción formaría un ciclo y no asignar el recurso
- Detectar: darse cuenta de que se formó un ciclo y tomar acciones
- Ignorar: no se hace nada

La más fácil de todas, y la que siguen la mayor parte de los sistemas operativos, es ignorar. Las otras posibilidades son muy costosas y los sistemas por lo general prefieren confiar en que los desarrolladores tendrán mucho cuidado al implementar sus aplicaciones. Por supuesto, nuestra experiencia demuestra que no siempre tienen la razón.

Actividad 1-11: Revise el siguiente código e identifique qué condición buscan evitar y qué desventajas tendría su utilización. Aquí, para reservar recursos (p.ej. semáforos) hay que reservarlos todos de manera "atómica"

```
P (mutex) ;  // empieza apropiación
  P (sem1) ;
  P (sem2) ;
  ...
V (mutex) ; // termina apropiación
```

1.7 Procesos

Continuemos formalizando el vocabulario. Hasta ahora en este documento hemos hablado indistintamente de procesos y threads. De hecho, el término thread apareció por el uso explícito que hacen de esa palabra las librerías de Java. Es hora de hacer la distinción porque se trata de dos conceptos bien diferentes.

Un proceso es un programa en ejecución. Y según dijimos esto significa que tiene un código y manipula un conjunto de datos. Tanto el código como los datos se deben encontrar en la memoria principal para poder ser ejecutados[3]. Adicional a esas dos porciones de memoria, debe existir un espacio para manejar la pila (donde guardaremos direcciones de retorno a medida que se llaman nuevos métodos/funciones) y debemos considerar que los registros del procesador también serán utilizados por un proceso. Por ejemplo, el registro PC (*Program Counter*) debe tener la dirección de la siguiente instrucción a ejecutar, el registro SP (*Stack Pointer*) debe tener la dirección del tope de la pila y los registros de trabajo deben tener los valores de los datos que están siendo accedidos por el proceso en un momento dado. Podemos representar un proceso gráficamente como lo ilustra la *Figura 1-2*.

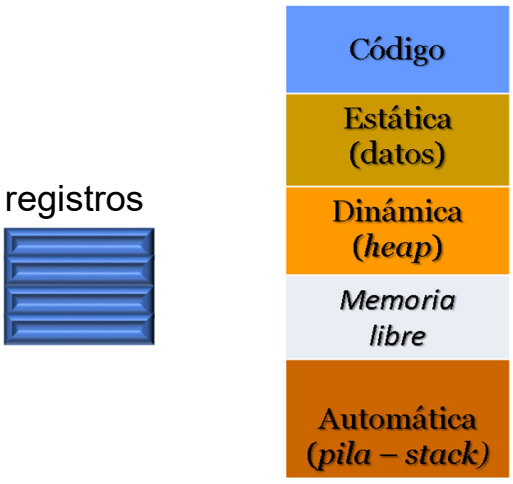

Figura 1-2 Componentes de un proceso

[3] Este es el concepto original definido en la Arquitectura Von Neumann

El sistema operativo además de asignar estos componentes de forma exclusiva a cada proceso, les asigna un identificador (PID - *Process Identifier*) y ofrece servicios para su manipulación. En particular, debe existir en el API del sistema operativo un servicio para crear un proceso. El proceso que invoque este servicio se conocerá como el padre del proceso creado.

La palabra en inglés para bifurcación es *fork*. Es por eso que este es el vocablo escogido por los sistemas operativos derivados de Unix (p.ej. Linux) y el estándar POSIX para identificar la rutina de creación de una copia de sí mismo por parte de un proceso. El proceso que invoca la rutina actuará como "proceso padre" y la nueva copia será el "proceso hijo". Los procesos resultantes son idénticos, excepto porque tienen PIDs distintos. En Unix, la llamada a *fork* retorna el PID del proceso hijo al padre y 0 al proceso hijo.

Crear una copia idéntica de un proceso significa hacer una copia (byte a byte) de todas las zonas de memoria del proceso, de los valores de los registros y de toda otra estructura de datos que utilice el sistema operativo para representar información sobre el proceso. Los procesos padre e hijo tienen cada uno su propia zona de memoria, pero el código es idéntico, los valores para los datos son los mismos, la siguiente instrucción a ejecutar (apuntada por el registro PC) es la misma, etc. Solo hay una diferencia y es en el valor de la variable que almacena el resultado de la rutina `fork()`. Para el padre, el valor de la variable será el PID del hijo y para el hijo el valor de la variable será 0.

fork nos permite entonces crear un proceso con una copia exacta del código y de los datos (y de la pila). Si se desea tener un proceso con código y datos diferentes, la manera de hacerlo es, una vez se ha creado la copia con `fork()`, se utiliza `exec()` que permite remplazar el código y datos actuales de un proceso. De esta manera se separan responsabilidades. Por un lado, se utiliza `fork()` para que el sistema operativo cree y actualice todas las estructuras necesarias para representar y ejecutar un proceso, y luego con `exec()` podemos lograr que un nuevo programa (disponible en un archivo ejecutable) se ejecute en el contexto de ese proceso ya creado. Como `exec()` no crea un proceso, se mantiene el PID del proceso que invoque este llamado al sistema, pero se remplazan el código, los datos, la pila y el heap por aquellos del nuevo programa.

En Unix existe una tercera primitiva asociada a `fork()` y `exec()`, se trata de `wait()`. `wait()`, como su nombre lo indica, se usa para que un proceso padre espere por la terminación de un hijo. Cuando un proceso llama a `wait()`, la siguiente instrucción de este proceso solo se ejecutará después de que el hijo cuyo PID es especificado como parámetro haya terminado su ejecución completamente. Utilizar `wait()` asegura una ejecución determinista entre el padre y el hijo.

En Windows la estrategia de creación es diferente. En este sistema operativo no se hace una copia, por lo tanto, la operación de creación (`CreateProcess`) realiza tanto la creación de las estructuras del sistema operativo como la copia en memoria del código y los datos a utilizar. Por eso esta rutina tiene muchos parámetros, 10 en total, empezando por el nombre del archivo que contiene el código (.exe).

Actividad 1-12: ¿Cuántos procesos, además del original, crea el siguiente programa?

```
for (i = 1; i < 2; i++)
    id = fork () ;
```

1.7.1 Threads

La gestión de procesos nos permite tener multiprogramación en un sistema. Un proceso ofrece un único punto de ejecución sobre el código (a partir del valor el registro PC) y asegura un concepto de encapsulación de todas las operaciones; cada zona de la memoria que utiliza un proceso es exclusiva para él. Si creamos otro proceso (incluso una copia como con `fork()`), obtenemos un espacio completamente independiente, no hay nada compartido entre un proceso padre y un proceso hijo. Sus zonas de código, datos, etc., son exclusivas de cada uno y por definición, un proceso no puede consultar y menos modificar un espacio de memoria que no le pertenece.

La programación multi-thread habilita tener más de un punto de ejecución sobre el mismo código, compartiendo el mismo espacio de direcciones (zonas de memoria). Un **thread** (hilo en español) es un proceso ligero o subproceso que se utiliza para tener flujos de ejecución independientes en el contexto de un único proceso. Un proceso puede tener varios hilos (mínimo uno) y todos ellos comparten las mismas zonas de código, datos, archivos abiertos, etc. Lo que es propio de cada hilo es su pila de ejecución y el estado de la CPU (incluyendo el valor de los registros). Un thread existe mientras exista el proceso que le brinda el contexto. Un proceso se ejecuta mientras quede al menos uno de sus threads en capacidad de ejecutarse en el sistema.

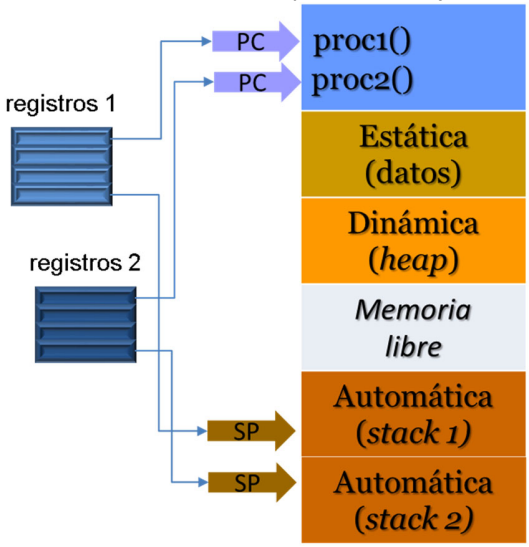

Figura 1-3 Un proceso con dos threads

El estándar POSIX define la rutina `pthread_create()` para la creación de un thread. La ejecución de un proceso empieza con la creación y ejecución de un thread inicial. A partir de ahí, el programador tiene la opción de crear más threads, dentro del mismo proceso, repitiendo este llamado al sistema. La librería `pthread.h` de Unix define un conjunto de funciones para la gestión (creación, comunicación y destrucción) y sincronización (exclusión mutua, señalamiento y encuentros) de threads. Un proceso es el contexto de los threads; en un sistema multi-thread los estados del sistema operativo (Listo, Ejecutado y Dormido) se aplican a los threads, no a los procesos, ya que en un mismo instante un proceso podría tener unos threads en Listo, uno (o más si hay varios procesadores) en Ejecutando y otros en Dormido.

La rutina `pthread_create()` retorna un entero para indicar el resultado de la creación (0 indica que fue exitoso). Recibe como parámetro un apuntador para retornar la identificación del thread creado, un apuntador

a una estructura con los parámetros de creación del thread, la rutina principal donde inicia la ejecución del thread y los parámetros de esta rutina. A continuación, presentamos un ejemplo de su uso en C.

```
void * funcion  ( void * p ) { ... }

pthread_t threadId [NTHREADS] ;
int id [NTHREADS] ;

for (i=0; i<NTHREADS; i++) {
    id [i] = i ;
    nok = pthread_create (&threadId [i], NULL, funcion,
                          (void *) &id [i]) ;
}
```

Actividad 1-13: Observe el uso del arreglo id. ¿Por qué es necesario usar un arreglo que lo único que contiene son los valores entre 0 y NTHREADS? ¿Qué pasa si pasamos &i como argumento?

En la misma librería `pthread.h` hay operaciones para manejar la sincronización entre threads. Se define el tipo `pthread_mutex_t` para declarar candados. Un candado es una compuerta que permite el acceso controlado a una zona de código, en ese sentido son iguales a los semáforos que se presentaron en la sección 1.5.2. Las operaciones `pthread_mutex_lock()` y `pthread_mutex_unlock()` sirven para proteger secciones críticas. La diferencia de estas operaciones con las que ya hemos visto es que estos candados tienen apropiación, es decir, solo el thread que ha realizado el *lock*, puede ejecutar el *unlock*. Las funciones `pthread_cond_wait()`, `pthread_cond_signal()` o `pthread_cond_broadcast()` implementan las funcionalidades de sincronización por eventos a nivel de threads. Por último, la rutina `pthread_join()` fuerza la espera de la finalización de un thread y la rutina `pthread_barrier_wait()` implementa la restricción de encuentro. El siguiente código ilustra el uso de `pthread_join()`. El thread inicial esperará que todos los threads creados terminen para continuar su ejecución.

```
for (i=0; i<NTHREADS; i++)
    pthread_create (&t [i], NULL, funcion, NULL) ;

for (i=0; i<NTHREADS; i++)
    pthread_join (t [i], NULL) ;
```

1.7.2 Comunicación de procesos

La comunicación entre threads es inmediata porque comparten la misma zona de datos, lo único que hay que hacer es ordenar el acceso por medio de los diferentes mecanismos de sincronización ofrecidos por el sistema operativo. Los procesos en cambio, si desean pasarse información entre ellos, tienen que recurrir a mecanismos de comunicación explícitos porque por definición no comparten nada en memoria principal (y acceder a memoria secundaria es muy costoso).

El mecanismo para comunicar procesos que no comparten espacios en memoria principal se conoce como paso de mensajes. Existen librerías que ofrecen la abstracción de memoria compartida sobre paso de mensajes, pero no son muy utilizadas por su bajo desempeño[4]. Las librerías de paso de mensajes, por su parte, habilitan la posibilidad de intercambiar información entre procesos que no comparten memoria principal. Hay dos maneras de hacer este intercambio: de manera directa utilizando la identificación de los procesos, o de manera indirecta utilizando una estructura intermedia que actúa como buzón entre las dos partes. El buzón tendrá un tamaño definido que determinará el máximo número (y tamaño) de los mensajes que se pueden almacenar en él sin que un proceso receptor los retire, pero si el tamaño del buzón es 0, al buzón se le conoce como canal y en este caso la comunicación debe ser sincrónica (el envío y la recepción son métodos que bloquean al proceso que los invoque hasta tanto la contraparte no ejecute la respectiva operación).

La comunicación entre procesos es un tema amplio que se extiende desde la comunicación entre threads de un mismo proceso (comunicación intra) hasta la comunicación entre procesos que se ejecutan en diferentes máquinas (comunicación inter), pasando por la comunicación entre procesos en la misma máquina. En cada caso tenemos posibilidades y restricciones distintas, por ello se requieren mecanismos diferentes. El alcance de este curso no incluye este tema que se aborda de manera más integral en los cursos de comunicaciones y de sistemas distribuidos.

1.7.3 Implementación de procesos

El sistema operativo gestiona el ciclo de vida de los procesos y eso implica su creación, despacho, destrucción y en general todos los servicios que pueda necesitar durante su ejecución: sincronización, comunicación, asignación de recursos, etc. El sistema operativo define el tiempo máximo de permanencia continua de un proceso en el estado Ejecutando (i.e. quantum), la memoria a la que tiene acceso exclusivo, los archivos que puede manipular, los dispositivos de Entrada/Salida que utiliza, etc.

La implementación del sistema operativo también se basa en un conjunto de procesos que comparten datos y que por lo tanto requieren de estos servicios y son objeto de todos los riesgos generados por la concurrencia. Particular atención se debe poner a los drivers de dispositivos porque es código que termina haciendo parte del sistema operativo y, si ese código no coopera adecuadamente con el código ya instalado, se pueden presentar los problemas de interbloqueos, inanición, etc., lo que a nivel de sistema operativo puede terminar en la necesidad de reiniciar el sistema completo.

Para la implementación de procesos, el sistema operativo utiliza una tabla, con una entrada por proceso, que contiene la información que debe mantener el sistema para asegurar la gestión a lo largo de la vida de cada proceso. Esto incluye el estado en el que se encuentra el proceso en un momento dado, el valor de los registros del procesador después de su última ejecución (esto incluye el registro PC, SP, etc. y todos los de trabajo donde se almacenan resultados temporales), información sobre los bloques de memoria en uso, el estado de los archivos que tenga abiertos el proceso, la información que se utilice a la hora de tomar decisiones para cambiar de estado (prioridades, señales que espera, etc.) y cualquier otra información que se requiera para asegurar que, cuando el proceso regrese al estado Ejecutando, lo podrá hacer en el punto exacto donde se encontraba la última vez que tuvo acceso al procesador. A esta estructura se le conoce como Bloque de Control de Proceso o **PCB** por sus siglas en inglés.

[4] Linda es un ejemplo de lenguaje de coordinación de procesos basado en la noción de tuplas que se pueden leer y escribir independiente de si los procesos comparten o no memoria.

Actividad 1-14: Lea las secciones 3.1 y 3.2 del libro de Silberschatz. En particular revise por qué es necesario tener dos nuevos estados (Nuevo y Terminado) para los procesos, la definición de interrupción, intercambio de contexto y planificación de procesos y la operación de la tabla de procesos del sistema operativo (PCB). ¿Cómo se altera el grafo de estados de un proceso en el caso de las máquinas con varios procesadores?

La gestión de los procesos es una de las tareas más delicadas, y que más influye en el desempeño del sistema, de un sistema operativo. La operación de retirar el procesador al proceso en Ejecutando y dárselo a uno de los procesos en el estado Listo se conoce como **cambio de contexto** y es una de las operaciones más críticas del sistema operativo. El cambio de contexto implica copiar desde los registros del procesador hacia el PCB del proceso que sale de Ejecutando, y desde el PCB del proceso seleccionado del estado Listo hacia los registros del procesador. Adicional a lo anterior, habrá que actualizar las colas afectadas y la información asociada al estado que se mantenga en el PCB. Estos movimientos de información consumen algún tiempo, tiempo en el cual el procesador no está ejecutando ningún proceso del usuario. Desde el punto de vista del usuario, este es un tiempo perdido del procesador. En general, cada vez que se ejecuta el sistema operativo, es tiempo de procesador que el usuario no puede aprovechar y que por lo tanto debe ser minimizado.

Además del costo mencionado en el párrafo anterior, cambiar la ejecución de un proceso a otro tiene más costos que es importante analizar. Al igual que un humano que está haciendo varias tareas al mismo tiempo, cuando cambia de tarea no solo debe guardar el estado de una tarea y recuperar el estado de otra, sino que le toma un tiempo adaptarse a la nueva rutina. En el caso de un proceso, la arquitectura de un procesador está diseñada para tener acceso rápido a los datos que un proceso utiliza frecuentemente (niveles de caché). Cambiar el contexto implica cambiar el código y/o los datos y ambas cosas se encuentran en memoria principal, por lo que, durante la ejecución de un proceso, el hardware se encarga que aquellas porciones de datos y código más utilizadas estén disponibles en la caché del procesador. Cuando cambiamos a un nuevo proceso, ese trabajo del hardware se pierde y debemos volver a empezar de nuevo a cargar en caché el código y/o datos más referenciados por el nuevo proceso. Dado que la eficiencia de las cachés es fundamental a la hora de mejorar el desempeño de un sistema, un cambio de contexto tiene un costo muy importante que el sistema operativo debe tener en cuenta.

Actividad 1-15: Si pudiera escoger al seleccionar el thread de la cola de Listo que debe pasar a Ejecutando, ¿prefería un thread del mismo proceso o un thread de un proceso distinto? ¿Cómo organizaría la asignación de threads a procesadores en una máquina multiprocesador (o multicore)? Evalúe los beneficios y las desventajas de su solución.

Lo anterior nos lleva a que la decisión de hacer un cambio de contexto debe ser bien analizada. Recordemos que la decisión de pasar de Listo a Ejecutando se da cuando el proceso en Ejecutando abandona este estado. Esto puede suceder porque el proceso termina, explícitamente solicita pasar a Dormido o simplemente se le termina su tiempo máximo de permanencia continua en este estado (quantum). Este tiempo máximo es un parámetro del sistema operativo en el que se deberá tener en cuenta el costo asociado a hacer el cambio. El sistema operativo buscará asegurar un equilibrio entre reactividad (los procesos en Listo necesitan acceder pronto al procesador) y desempeño (limitar el tiempo perdido de procesador debido a las transiciones de estado y cambios de contexto).

Que una de las razones para ejecutar un cambio de contexto sea el tiempo, implica que el procesador debe tener esa noción incorporada de alguna manera. El hardware (y luego el sistema operativo) necesita algún referente

de tiempo y de paso un mecanismo que le permita retomar el control de la ejecución que sucede en el procesador. De otra manera, mientras un proceso esté dentro de su quantum, las únicas instrucciones que se podrían ejecutar son las de ese proceso. Pero ¿y si sucede algo extraordinario? O si simplemente se acabó el tiempo, ¿cómo retomamos el control de la máquina?

Un procesador tiene múltiples líneas de entrada y de salida para comunicarse con el resto del hardware de un computador. Una de esas líneas de entrada se conoce como INT y se utiliza para **interrumpir** al procesador en cualquier momento en que se requiera su atención. Cuando un dispositivo requiere algo urgente del procesador, enviará una señal por esa línea, lo que tendrá como consecuencia que el procesador inicie un mecanismo de atención urgente a quien lo solicita. Más adelante entraremos a analizar en detalle esta operación, por ahora concentrémonos en que a través de esta señal el componente reloj físico de la maquina envía una señal cada X milisegundos para indicar justamente eso: que han pasado X ms desde la última vez que envió una señal por esta línea. Esta advertencia, que debe llegar hasta el sistema operativo, será utilizada para determinar la necesidad de realizar un cambio de contexto. Si el sistema operativo determina que se ha agotado el quantum del proceso actualmente en ejecución (quantum que debe ser un múltiplo de X ms), procederá a realizar el cambio de contexto.

Actividad 1-16: Busque qué valores se manejan típicamente para X y para el quantum. De ¿qué cree que dependen estos valores?

Actividad 1-17: Lea el capítulo 4 sobre Threads del libro de R. Garg y G. Verma. Con menos detalle también lo encuentra en el libro de Silberschatz, sección Multithreading Model (4.2 o 4.3 según la versión). Investigue ¿Qué tipo de threads son los de Java?

1.7.4 Procesos y threads en Linux

Linux tiene un modelo muy flexible que no se limita a las dos posibilidades que hemos visto de procesos y threads. Las unidades de ejecución en Linux se conocen como **tareas** (*tasks*). Las tareas pueden compartir distintos recursos entre ellas y dependiendo de qué compartan se podrán asemejar a procesos o a threads (cualquiera de los dos). La creación de una tarea se hace llamando a la primitiva `clone()` del sistema operativo, especificando el nivel de compartición deseado.

Si al crear una tarea se especifica que no se desea compartir nada, obtendremos un proceso completamente independiente. Si por el contrario se define que se desea compartir el código, los datos y todos los elementos disponibles, obtendremos una unidad de ejecución que comparte el contexto completo de otro, y de acuerdo con lo que hemos presentado, esta es la definición de un thread. Linux mantiene una estructura para cada tarea que apunta a los diferentes elementos de su contexto, al ejecutar `clone()` se crea una nueva estructura que puede apuntar a los mismos o diferentes elementos de la estructura inicial.

1.7.5 Implementación de la sincronización

Vamos a terminar esta presentación sobre procesos desvelando el último de los misterios sobre la manera como se implementa la sincronización entre procesos. Hasta ahora hemos descargado la responsabilidad en el sistema

operativo y su capacidad de ofrecer semáforos con primitivas *atómicas* para su manipulación. Pero, ¿cómo hace el sistema operativo para ofrecer servicios que aseguren que no son interrumpidos por el hardware entre dos puntos de código?

Las interrupciones son un mecanismo poderoso del hardware para recuperar el control ante cualquier eventualidad. Es gracias a las interrupciones que, cuando se ha cumplido el tiempo máximo en el estado Listo, se inicia un cambio de contexto que lleva a que el proceso en el estado Ejecutando pierda el procesador para cedérselo a uno de los procesos en estado Listo. Cuando introducimos la atomicidad de las operaciones P() y V() de los semáforos, veíamos que estas operaciones están en la base de construcciones más elaboradas de sincronización y que era un requisito el que fueran atómicas para su correcta operación. Si P() o V() pudieran ser interrumpidas, no proveerían garantía sobre la coherencia en el acceso al contador o la cola en su estructura.

Una estrategia posible para el sistema operativo es deshabilitar las interrupciones durante la ejecución de estas rutinas especiales. Esta solución, que ofrecen la mayoría de los procesadores, hace que el procesador deje de escuchar señales por la señal INT y, por supuesto, solo está disponible para el sistema operativo. Cualquier otro programa que intente dar esta orden al procesador generará un error por el peligro que ella supone. El sistema operativo por su parte, consciente del riesgo, puede asegurar que no se causen problemas por inhabilitar las interrupciones por un periodo de tiempo muy corto. Una vez terminada de ejecutar la sección crítica, el sistema operativo vuelve a habilitar las interrupciones y todo continúa normalmente.

Actividad 1-18: Piense un ejemplo concreto de un problema que podría presentarse si se permite que un proceso normal deshabilite las interrupciones.

Una solución que no requiere de deshabilitar es usar la instrucción XCHG (*assembler*). Deshabilitar las interrupciones permite que modifiquemos dos o más valores con la seguridad de no perder el procesador entre una modificación y otra. XCHG modifica dos valores en una sola instrucción, con lo cual las dos operaciones se ejecutan de forma atómica (recuerde que las interrupciones solo son atendidas después de terminar de ejecutar la instrucción de hardware actual). Esta instrucción intercambia el valor de sus dos operandos que pueden ser ambos registros del procesador o uno de ellos ser una dirección en memoria. Esta propiedad es muy especial porque permite consultar un valor y modificarlo, todo en la misma operación, de manera que después de modificarlo todavía puedo saber qué valor tenía antes de la modificación.

La instrucción XCHG está en la base de la implementación de las clases atómicas de Java, en particular es la manera de lograr que operaciones de getAndSet() sean atómicas. Esta instrucción realiza una reserva implícita del bus de comunicaciones hacia la memoria lo que evita que cualquier otro procesador pueda manipular datos en RAM mientras se ejecuta XCHG. Esta capacidad asegura que XCHG funcione incluso en arquitecturas multiprocesador, a diferencia de la solución de deshabilitar interrupciones que funciona para un solo procesador.

1.8 Programación asincrónica

La programación asincrónica es una característica muy útil que se habilita por la posibilidad de tener varios flujos de ejecución activos en una máquina.

Asincronía: se refiere a sucesos que no tienen lugar en total correspondencia temporal entre ellos

La programación asincrónica nos da la capacidad de mantener un flujo de ejecución avanzando, mientras otros flujos también lo hacen. Nos ofrece entonces, por ejemplo, la posibilidad de solicitar un elemento y no tener que esperar pasivamente a que el elemento solicitado esté disponible para poder continuar, sino que, por el contrario, podemos seguir ejecutando otras cosas y, en el momento adecuado, ser avisados de la disponibilidad de lo esperado. Un ejemplo sencillo de lo anterior sería cuando usted quiere descargar dos películas (de las cuales tiene las licencias correspondientes por supuesto). Podría hacerlo sincrónicamente, una después de otra, esperando a que termine la primera descarga para comenzar la segunda, con un tiempo total de la descarga igual a la sumatoria del tiempo de la descarga de la una más el de la otra. O podría hacerlo asincrónicamente, solicitar la descarga de ambas películas al tiempo y ser avisado cuando cada descarga esté lista, obteniendo un tiempo total de descarga que será el mayor de los dos tiempos de descarga.

En APO1 ya habíamos usado la programación asincrónica. Una ventana que tiene varios botones no obliga a procesar las acciones del usuario una detrás de la otra. Por el contrario, desplegamos todos los botones en la ventana y vamos respondiendo a medida que el usuario va seleccionando acciones. De hecho, dependiendo de cómo esté hecho el código, mientras el programa responde a una solicitud del usuario, este podría hacer nuevas solicitudes a la aplicación.

Java ofrece 12 eventos distintos a los que podemos asociarles acciones para que sucedan de manera asincrónica con el resto de la ejecución de una aplicación. En APO1 ustedes utilizaron el tipo ActionEvent para detectar y responder a la acción de oprimir un botón de la interfaz, pero hay más eventos.

Los eventos en Java son objetos creados como resultado de las acciones del usuario en la interfaz. Si el usuario oprime un botón, selecciona algo en un combo box o escribe en una caja de texto, el sistema lanza un evento que se convierte en un objeto que debe ser procesado por el manejador de eventos de Java, implementado en la biblioteca Swing. Cada tipo de evento tiene su propio manejador que debe ser implementado para definir las acciones a ejecutar cuando suceda el evento. Estas acciones se ejecutan de manera asincrónica con el resto de aplicaciones que se estén ejecutando en el sistema (incluida la propia de la cual hacen parte estos eventos).

El manejo de los periféricos de un computador sigue el mismo modelo. Cuando un dispositivo requiere atención del procesador central, usa el mecanismo de interrupciones para avisar que requiere atención y el sistema operativo ejecutará asincrónicamente con el resto de actividades en el computador las acciones necesarias para atender el requerimiento del dispositivo. En esta sección vamos a estudiar cómo es esa interacción entre los dispositivos y el procesador central.

Lo primero que tenemos que entender es cómo son físicamente estos dispositivos. Casi todos responden al mismo modelo, un dispositivo mecano-electrónico, un controlador que conoce el dispositivo y que por lo tanto le puede dar órdenes y recibir respuestas y una interfaz para conectarlo al bus principal del computador. La *Figura 1-4* presenta la estructura general de un dispositivo. Los registros de la interfaz serán el mecanismo para pasar instrucciones desde el procesador hacia el controlador del dispositivo o resultados desde el controlador hacia el procesador.

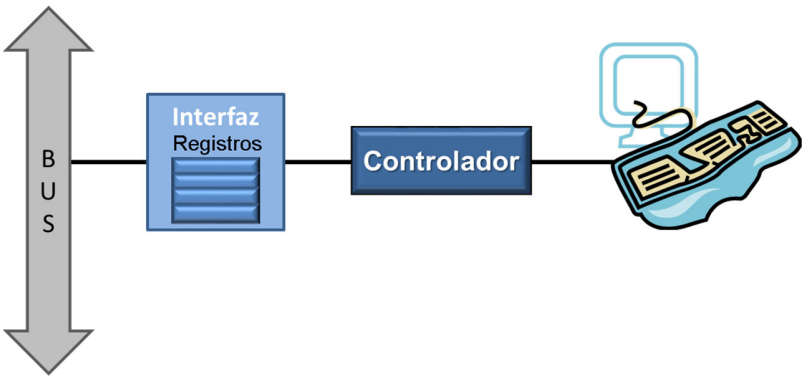

Figura 1-4 Estructura de un dispositivo

Una vez acordada la sintaxis y semántica entre el procesador y el controlador, quedan tres problemas por resolver en esta interacción: i) ¿Qué direcciones utilizar para hacer referencia a los registros de la interfaz, de manera que no se confundan con las direcciones de la memoria principal? ii) ¿Cómo transferir un gran conjunto de datos entre la memoria y los dispositivos? Y iii) ¿Cómo identificar el dispositivo que requiere atención en un momento dado? Las soluciones que presentaremos ofrecen alternativas para resolver estas problemáticas.

El primer problema tiene que ver con cómo comunicar el procesador central y los periféricos. Recuerde de su curso de Fundamentos de Infraestructura de Tecnología que el procesador se comunica con la memoria a través del bus de comunicaciones, que a su vez se divide en bus direcciones, bus de datos y bus de control. El procesador pone en el bus de direcciones la dirección que quiere leer o escribir de/en la memoria, y la memoria al identificar esa dirección pone en el bus de datos el contenido de la posición de memoria especificada (lectura) o toma del bus de datos un valor y lo escribe en la posición de memoria (escritura). La especificación de si se trata de una operación de lectura o una de escritura se hace por medio del bus de control. Cada periférico tendrá su propio conjunto de registros en la interfaz y necesitaremos asignarles una dirección para que el procesador puede leer o escribir en ellos. Existen dos alternativas:

Mapear direcciones de memoria: en este caso lo que hacemos es “robarnos” unas direcciones de memoria y decir que esas direcciones no se utilizan para la RAM sino para dispositivos. De esa manera, la CPU genera las direcciones de manera normal y dependiendo del rango, estas pueden estar haciendo referencia a una posición en RAM o a un registro en la interfaz de un dispositivo.

Mapear puertos o utilizar direcciones propias: este mecanismo de comunicación crea un espacio de direcciones propio para los dispositivos. De esta manera una dirección N ahora puede hacer referencia a una dirección en memoria o a un registro de un dispositivo. Para poder diferenciar a qué tipo corresponde una dirección, se requiere de una línea adicional en el bus de control que permita saber si el valor en el bus de direcciones se refiere a una posición en memoria o no (caso en el cual se trata de un registro). Dicha línea se conoce como M/IO y su activación o no depende de la instrucción utilizada por el procesador al momento de generar la dirección. En la arquitectura 8086, la instrucción MOV pone la línea M/IO en 0, indicando que se trata de un acceso a memoria, mientras que las instrucciones IN y OUT ponen esa línea en 1, lo que significa que la dirección debe ser ignorada por la memoria y que quien debe actuar es el periférico asociado a esa dirección.

Una vez resuelto el direccionamiento, debemos ocuparnos de la transferencia. Los datos se transfieren entre la memoria principal y los dispositivos de Entrada/Salida (E/S), dos elementos que manejan velocidades eventualmente muy diferentes. Por ello existen dos maneras de transferir datos:

Transferencia programada: se utiliza principalmente cuando el tamaño de los datos a transferir es muy pequeño. Piense por ejemplo en la transmisión entre un teclado y el procesador. En este caso, cada vez que se oprime la tecla se debe transferir el caracter, ya que queremos que cada vez que tecleemos algo, inmediatamente lo veamos reflejado en la pantalla. Como la transferencia es tan pequeña tiene sentido distraer al procesador unos pocos microsegundos para hacerla. Por muy rápido que tecleemos, la distracción va a ser mínima. El procesador podría copiar el dato desde el registro del dispositivo hasta alguno de sus registros y, si es necesario, transferirlo luego a una posición de memoria.

Acceso Directo a memoria (*Direct Memory Access* - DMA): utilizar el procesador central para transferir unos pocos bytes tiene sentido porque no afectamos sensiblemente el desempeño de las aplicaciones. Pero si quisiéramos transferir varios Megas o Gigas de información, esto consumiría mucho tiempo del procesador y la afectación sería más que perceptible. En estos casos se prefiere no utilizar el procesador principal y ponerle a la máquina (o a cada dispositivo) un procesador más pequeño (menos potente y más barato) para que se ocupe de estas transferencias. De esta manera, el procesador central solo se ocupa de ordenarle la transferencia a este procesador de DMA, y él se ocupará de la transferencia entre la zona de memoria y el periférico.

La última problemática en la interacción entre los dispositivos y el procesador central tiene que ver con la manera que tiene el procesador central para identificar el dispositivo que solicita atención. Cuando un proceso solicita una operación de E/S es muy probable que el sistema operativo decida pasar este proceso al estado Dormido porque la operación se va a demorar mucho. El sistema operativo recibe la solicitud de E/S y ordena al periférico en cuestión realizarla, pero debido a las diferencias de velocidades, lo más rentable es seleccionar un nuevo proceso de Listo y asignarle a él el procesador mientras tanto. Cuando el periférico que debía realizar la E/S termina, este debe avisar que ya terminó y que por lo tanto se puede pasar al proceso solicitante del estado Dormido al estado Listo. Pero como puede haber varias operaciones de E/S en curso, es necesario identificar cuál es el periférico que terminó, esto es, cuál necesita atención. De nuevo, existen dos maneras de hacerlo:

Sondeo (polling): en esta estrategia, una vez el procesador central detecta que un dispositivo requiere atención, hace un sondeo, uno a uno de los dispositivos, preguntando a cada uno si es ese dispositivo el que necesita alguna acción por parte del sistema operativo. Por supuesto, si el número de dispositivos es muy alto, se podría desperdiciar mucho tiempo del procesador en este sondeo.

Vector de interrupciones: para evitar preguntar a todos y cada uno de los dispositivos, se puede establecer un protocolo en el cual un dispositivo que requiere atención, además de generar una interrupción, pone alguna identificación en el bus de datos. La manera eficiente que se tiene para esto es que, dado que cada dispositivo es único y requiere de rutinas muy particulares para atenderlo, las direcciones de dichas rutinas, que hacen parte del sistema operativo, se dejan en una zona especial de la memoria, llamada vector de interrupciones. Cuando un dispositivo necesita ejecutar su rutina de atención, lo que hace es interrumpir (asincrónicamente) al procesador, por medio de la línea INT, y una vez el procesador responde que reconoce la interrupción, el controlador del dispositivo copia al bus de datos la entrada en el vector en donde se encuentra la rutina de tratamiento asociada al dispositivo. El procesador (por hardware como reacción a la señal detectada en la línea INT) lee esta dirección y la coloca en el registro PC (Contador de Programa) lo que causa que automáticamente la ejecución pase a la rutina apropiada.

Terminaremos esta sección por aclarar el rol de los **drivers** en la operación de un dispositivo. El driver es el componente software de un dispositivo que hace parte del sistema operativo y que por lo tanto se ejecuta en un modo de altos privilegios. A pesar de ser parte de sistema operativo, es software escrito por terceros, que se ejecuta concurrentemente con otros servicios del sistema operativo, lo que lo hace susceptible a todas las problemáticas que hablamos en secciones anteriores de este documento. Todo el código dependiente del dispositivo se encuentra en el driver, por esta razón, existe un driver por dispositivo en el sistema, o al menos un driver por tipo de dispositivo en el caso de los drivers genéricos, que, si bien pueden funcionar, puede que no incluyan mecanismos para utilizar todas las funcionalidades de un dispositivo particular.

El código dependiente tiene dos partes: las instrucciones para solicitarle algo al dispositivo y las instrucciones a ejecutar cuando el dispositivo termina una solicitud o requiere atención inmediata (porque por ejemplo el usuario realizó una acción particular). El código para enviar solicitudes al dispositivo será llamado desde las rutinas de entrada/salida que invoca el desarrollador, y el código a ejecutar cuando el dispositivo necesita entregar información al sistema operativo es justo la rutina de la cual instalaremos su dirección en el vector de interrupciones.

1.9 Ejemplo práctico: servidores concurrentes

La última sección de este capítulo la dedicaremos a estudiar una implementación que se ha convertido, más que en un ejemplo, en un patrón arquitectural utilizado por muchas aplicaciones: cliente servidor, el utilizado por la web. La web es una aplicación distribuida que permite que una aplicación que se ejecuta en una máquina (cliente o navegador) consulte, transfiera o manipule, información de una aplicación que se ejecuta en otra máquina (servidor web). La manera de conectar las dos aplicaciones es a través del protocolo HTTP pero este es un tema del curso de Infraestructura de Comunicaciones. En este curso nos vamos a ocupar de entender cómo es la arquitectura del servidor Web, pero para ello, primero vamos a presentar algunos conceptos básicos de la comunicación entre procesos que se ejecutan en diferentes máquinas.

Para lograr comunicar dos procesos en máquinas distintas lo primero que necesitamos es una manera de identificar la otra máquina y designar un proceso particular en ella. Las direcciones IP cumplen con la primera función. Toda máquina conectada a Internet tiene una única dirección en el mundo. Usted puede consultar la suya visitando la página https://whatismyipaddress.com en su navegador. En su caso (cliente) esta dirección puede cambiar de un día para otro. Pero en el caso de alguien que quiere ofrecer un servicio (servidor web) esa dirección debería ser fija y no debería cambiar. Como consecuencia, para que haya comunicación entre un cliente y un servidor, el primero debe conocer la dirección del segundo. Además, el cliente siempre debe iniciar la comunicación.

Pero la dirección IP no es suficiente porque un cliente no habla con una máquina, habla con un proceso particular de esa máquina. Para establecer el proceso con el que se quiere comunicar en una máquina remota, en lugar de intentar averiguar el identificador del proceso remoto, lo que se hace es un acuerdo entre las dos partes y se establece un número de puerto por el cual se van a comunicar. Un puerto no es más que un número de buzón en una máquina. Cada proceso se suscribe a un buzón en su máquina local y todo lo que llegue a ese buzón le será entregado a ese proceso. Solo puede haber un proceso suscrito a un buzón en un momento dado. Una máquina tiene 64K buzones. Una misma aplicación puede usar varios buzones a la vez.

El sistema operativo implementa un servicio llamado sockets para simplificar el proceso de envío y recepción de mensajes entre procesos en distintas máquinas. Para comunicarse con un proceso remoto en otra máquina, un proceso local debe primero solicitar al sistema operativo la creación de un socket (especificando la IP y el puerto en ambas máquinas). Una vez el socket está establecido, el proceso solo debe escribir en el socket para enviar un mensaje y leer del socket para recibir un mensaje. Tanto el envío como la recepción son sincrónicos lo que quiere decir que, si un proceso envía un mensaje a través de un socket, esa acción solo termina una vez el proceso correspondiente ha recibido el mensaje. Igualmente, si un proceso lee de un socket donde no hay información, pasará al estado Dormido, esperando el evento asociado con la escritura en ese socket.

Una implementación de un servidor web como un **servidor iterativo** tendría el siguiente comportamiento: el servidor esperaría la solicitud de conexión de un cliente, si no hay clientes el servidor esperaría Dormido a que uno se conecte. Cuando un cliente se conecta, se crea un socket y el servidor leería y escribiría sobre ese socket para atender las solicitudes del cliente. Luego de atender al cliente, el servidor repetiría el procedimiento en un ciclo infinito. El comportamiento descrito anteriormente explica por qué es necesario implementar un servidor web como una aplicación concurrente. Un servidor web debe recibir peticiones de múltiples clientes y debe atenderlos a todos al tiempo; debe leer solicitud de todos los clientes, pero no puede bloquearse leyendo de un socket si una solicitud no está lista. Un servidor debe estar en capacidad de leer de varios sockets al mismo tiempo, algo que solo se puede hacer con una implementación concurrente.

Existen 3 estrategias claramente identificables para la implementación concurrente de un servidor web. La primera es implementar un servidor multi-thread, y crear un thread cada vez que llega una nueva solicitud. De esta manera, cada vez que un cliente hace una solicitud, se crea un thread para atenderla y ese thread se encarga de armar la respuesta y enviarla al cliente originador de la solicitud. Esta solución, **thread por solicitud**, permite que toda nueva solicitud sea atendida independientemente de lo que suceda con otras solicitudes.

Una segunda posibilidad es tener un **thread por conexión**. En este caso se crea un thread cada vez que llega un nuevo cliente a hacer solicitudes al servidor. Pero esta vez, todas las solicitudes del mismo cliente son recibidas, procesadas y contestadas por el mismo thread servidor. La idea de fondo del thread por conexión es disminuir el número de threads creados por un servidor y así limitar los llamados al sistema por ese servicio. Recuerde que la operación de crear un thread tiene un costo no despreciable y si un mismo cliente realiza varias solicitudes, una detrás de otra, atenderlas con el mismo thread puede ser mucho más eficiente.

Por supuesto, soportar concurrencia con threads no es la única opción. Uno podría imaginarse que en lugar de threads se crean procesos por conexión o por solicitud, sin embargo, esa posibilidad es demasiado costosa y en la práctica no se utiliza. Incluso, crear threads bajo demanda también es una operación pesada y con un riesgo importante de agotar recursos si no se controla. Por eso la mayoría de las implementaciones lo que hacen es que preparan un **pool de threads** que tienen disponible desde el momento en que arranca el servidor. La ventaja de esta aproximación es que no hay que crear threads en el momento en que llega el cliente o su solicitud con lo cual la respuesta es más rápida y además se evita la destrucción y creación continua de threads durante la operación del servidor. Otra ventaja de tener definido el pool con anterioridad es que no hay riesgo de agotar los recursos de la máquina ante un eventual incremento de la demanda. Los ataques de Negación de Servicio típicamente buscan agotar los recursos del servidor, por eso, mantener controlado el número de threads que se van a activar en un servidor, limita el impacto de un ataque. La desventaja de esta solución es la dificultad de estimar correctamente este número de threads para el funcionamiento adecuado del servidor. Si estimamos por encima, tendríamos un desperdicio de recursos; si estimamos por debajo, una petición/conexión podría tener que esperar para ser atendida.