Universidad De Granada

Grado ingeniería informática (2016 – 2017)

Sistemas Operativos

Hebras | Hilos | Threads

Trabajo realizado por Antonio Miguel Morillo Chica para María Angustias Sánchez Buendía

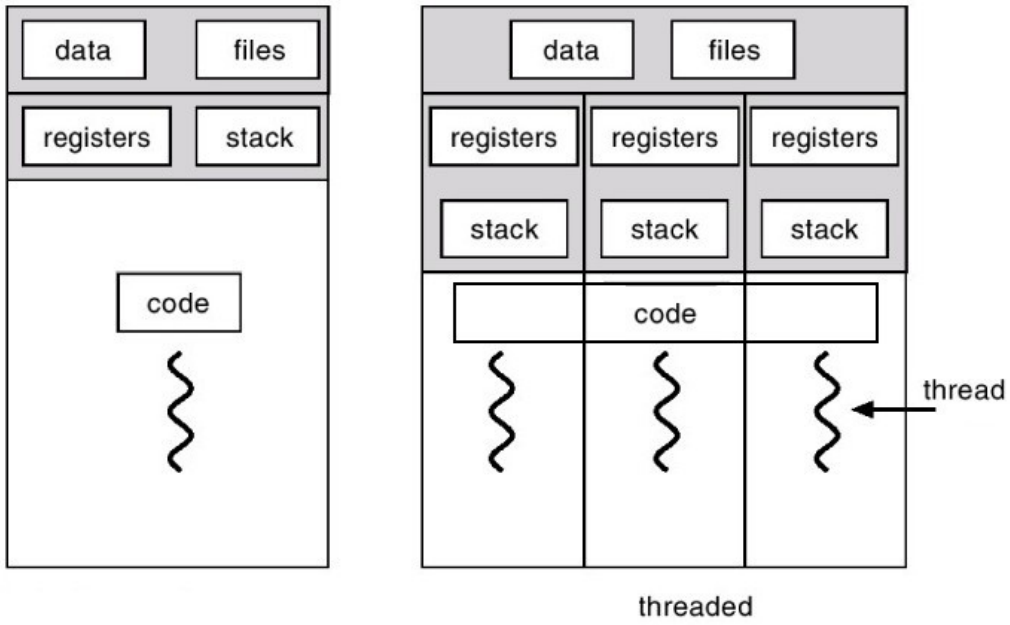
---

Índice

1. Introducción.

Un hilo de ejecución, en los sistemas operativos, es similar a un proceso en que ambos representan una secuencia simple de instrucciones ejecutada en paralelo con otras secuencias aunque esto no es siempre así porque podremos hablar de paralelismo o concurrencia entre procesos/hilos. Los hilos permiten dividir un programa en dos o más tareas que corren simultáneamente, por medio de la multiprogramación. En realidad, este método permite incrementar el rendimiento de un procesador de manera considerable. En todos los sistemas de hoy en día los hilos son utilizados para simplificar la estructura de un programa que lleva a cabo diferentes funciones.

Todos los hilos de un proceso comparten los recursos del proceso. Residen en el mismo espacio de direcciones y tienen acceso a los mismos datos. Cuando un hilo modifica un dato en la memoria, los otros hilos utilizan el resultado cuando acceden al dato. Cada hilo tiene su propio estado, su propio contador, su propia pila y su propia copia de los registros de la CPU. Los valores comunes se guardan en el bloque de control de proceso (PCB), y los valores propios en el bloque de control de hilo (TCB).



Muchos lenguaje de programación (como Java), y otros entornos de desarrollo soportan los llamados hilos o hebras (en inglés, threads). Un ejemplo de la utilización de hilos es tener un hilo atento a la interfaz gráfica (iconos, botones, ventanas), mientras otro hilo hace una larga operación internamente. De esta manera el programa responde más ágilmente a la interacción con el usuario.

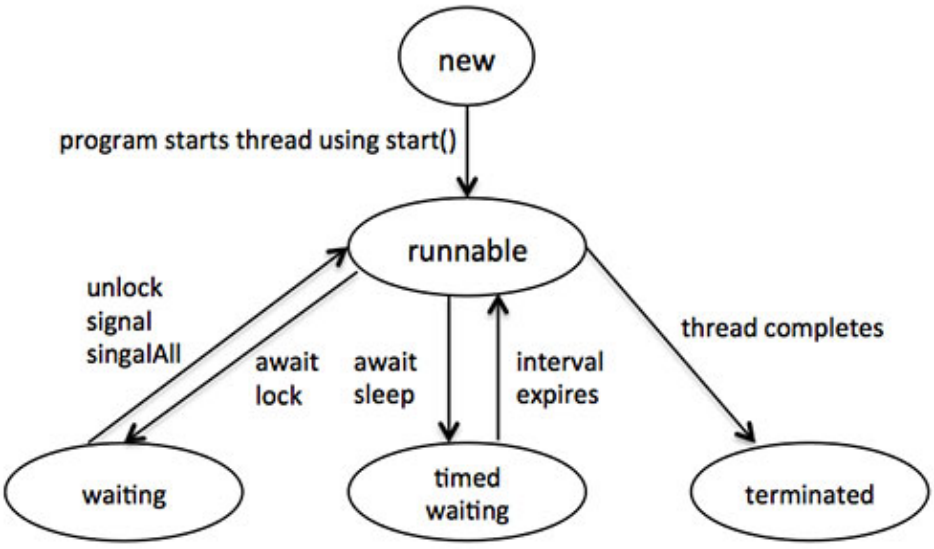
1.2. Diferencias entre proceso e hilo.

Los hilos se distinguen de los tradicionales procesos en que los procesos son generalmente independientes, llevan bastante información de estados, e interactúan sólo a través de mecanismos de comunicación dados por el sistema. Por otra parte, muchos hilos generalmente comparten otros recursos directamente. En los sistemas operativos que proveen facilidades para los hilos, es más rápido cambiar de un hilo a otro dentro del mismo proceso, que cambiar de un proceso a otro. Este fenómeno se debe a que los hilos comparten datos y espacios de direcciones, mientras que los procesos al ser independientes no lo hacen. Al cambiar de un proceso a otro el sistema operativo (mediante el dispacher) genera lo que se conoce como overhead, que es tiempo desperdiciado por el procesador para realizar un cambio de modo (mode switch), en este caso pasar del estado de Runnig al estado de Waiting o Bloqueado y colocar el nuevo proceso en Running. En los hilos como pertenecen a un mismo proceso al realizar un cambio de hilo este overhead es casi despreciable.

1.3. Funcionalidad de los hilos.

Al igual que los procesos, los hilos poseen un estado de ejecución y pueden sincronizarse entre ellos para evitar problemas de compartimiento de recursos. Generalmente, cada hilo tiene especificada una tarea específica y determinada, como forma de aumentar la eficiencia del uso del procesador. Los principales estados de ejecución de los hilos son: Ejecución, Listo y Bloqueado. No tiene sentido asociar estados de suspensión de hilos ya que es un concepto de proceso. En todo caso, si un proceso está expulsado de la memoria principal (ram), todos sus hilos deberán estarlo ya que todos comparten el espacio de direcciones del proceso. Las transiciones entre estados más comunes son las siguientes:

- Creación: Cuando se crea un proceso se crea un hilo para ese proceso. Luego, este hilo puede crear otros hilos dentro del mismo proceso. El hilo tendrá su propio contexto y su propio espacio de pila, y pasara a la cola de listos.
- Bloqueo: Cuando un hilo necesita esperar por un suceso, se bloquea (salvando sus registros). Ahora el procesador podrá pasar a ejecutar otro hilo que este en la cola de Listos mientras el anterior permanece bloqueado.
- Desbloqueo: Cuando el suceso por el que el hilo se bloqueo se produce, el mismo pasa a la cola de Listos.
- Terminación: Cuando un hilo finaliza se liberan tanto su contexto como sus pilas.



1.4. Ventajas de los hilos frente a los procesos.

Si bien los hilos son creados a partir de la creación de un proceso, podemos decir que un proceso es un hilo de ejecución, conocido como monohilo. Pero las ventajas de los hilos se dan cuando hablamos de Multihilos, un procesos tiene múltiples hilos de ejecución los cuales realizan actividades distintas, que pueden o

no ser cooperativas entre si. Los beneficios de los hilos se derivan de las implicaciones de rendimiento y se pueden resumir en los siguientes puntos:

- Se tarda mucho menos tiempo en crear un hilo nuevo en un proceso existente que en crear un proceso. Algunas investigaciones llevan al resultado que esto es así en un factor de 10.
- Se tarda mucho menos en terminar un hilo que un proceso, ya que su cuando se elimina un proceso se debe eliminar el PCB del mismo, mientras que un hilo se elimina su contexto y pila.
- Se tarda mucho menos tiempo en cambiar entre dos hilos de un mismo proceso.
- Los hilos aumentan la eficiencia de la comunicación entre programas en ejecución. En la mayoría de los sistemas en la comunicación entre procesos debe intervenir el núcleo para ofrecer protección de los recursos y realizar la comunicación misma. En cambio, entre hilos pueden comunicarse entre si sin la invocación al núcleo. Por lo tanto, si hay una aplicación que debe implementarse como un conjunto de unidades de ejecución relacionadas, es más eficiente hacerlo con una colección de hilos que con una colección de procesos separados.

2. Aspectos de la ejecución.

Hay una serie de aspectos a la hora de implementar un sistema de gestión de hilos muy críticos como puede ser la sincronización entre hilos y que en algunos casos son dependientes del hardware que se este usando o bien de la plataforma de desarrollo (que versión de la maquina virtual Java se esta usando por ejemplo) o incluso del propio código que los desarrolladores de software crean.

2.1 Sincronización de hilos.

Todos los hilos comparten el mismo espacio de direcciones y otros recursos como pueden ser archivos abiertos. Cualquier modificación de un recurso desde un hilo afecta al entorno del resto de los hilos del mismo proceso. Por lo tanto, es

necesario sincronizar la actividad de los distintos hilos para que no interfieran unos con otros o corrompan estructuras de datos.

Una ventaja de la programación multihilo es que los programas operan con mayor velocidad en sistemas de computadores con múltiples CPUs (sistemas multiprocesador o a través del grupo de máquinas) ya que los hilos del programa se prestan verdaderamente para la ejecución concurrente. En tal caso el programador necesita ser cuidadoso para evitar condiciones de carrera (problema que sucede cuando diferentes hilos o procesos alteran datos que otros también están usando), y otros comportamientos no intuitivos. Los hilos generalmente requieren reunirse para procesar los datos en el orden correcto. Es posible que los hilos requieran de operaciones atómicas para impedir que los datos comunes sean cambiados o leídos mientras estén siendo modificados, para lo que usualmente se utilizan los semáforos. El descuido de esto puede generar interbloqueo.

2.2 Formas de los multihilos.

Los Sistemas Operativos generalmente implementan hilos de dos maneras:

- Multihilo apropiativo: Permite al sistema operativo determinar cuándo debe haber un cambio de contexto. La desventaja de esto es que el sistema puede hacer un cambio de contexto en un momento inadecuado, causando un fenómeno conocido como inversión de prioridades y otros problemas.
- Multihilo cooperativo: Depende del mismo hilo abandonar el control cuando llega a un punto de detención, lo cual puede traer problemas cuando el hilo espera la disponibilidad de un recurso.

El soporte de hardware para multihilo desde hace poco se encuentra disponible. Esta característica fue introducida por Intel en el Pentium 4, bajo el nombre de HyperThreading.

2.3 Uso de los multihilos.

Algunos de los casos más claros en los cuales el uso de hilos se convierte en una tarea fundamental para el desarrollo de aplicaciones son:

- Trabajo interactivo y en segundo plano: por ejemplo, en un programa de hoja de cálculo un hilo puede estar visualizando los menús y leer la entrada del usuario mientras que otro hilo ejecuta las órdenes y actualiza la hoja de cálculo. Esta medida suele aumentar la velocidad que se percibe en la aplicación, permitiendo que el programa pida la orden siguiente antes de terminar la anterior.
- Procesamiento asíncrono: los elementos asíncronos de un programa se pueden implementar como hilos. Un ejemplo es como los software de procesamiento de texto guardan archivos temporales cuando se esta trabajando en dicho programa. Se crea un hilo que tiene como función guardar una copia de respaldo mientras se continúa con la operación de escritura por el usuario sin interferir en la misma.
- Aceleración de la ejecución: se pueden ejecutar, por ejemplo, un lote mientras otro hilo lee el lote siguiente de un dispositivo.
- Estructuración modular de los programas: puede ser un mecanismo eficiente para un programa que ejecuta una gran variedad de actividades, teniendo las mismas bien separadas mediante a hilos que realizan cada una de ellas.

3. Implementación de los hilos.

Hay dos grandes categorías en la implementación de hilos:

- Hilos a nivel de usuario, ULT (User Level Thread)
- Hilos a nivel de Kernel KLT (Kernel Level Thread)

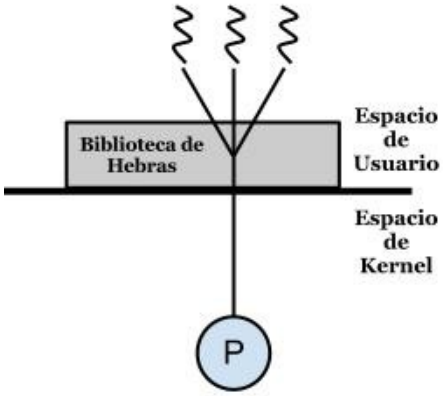
Por otro lado encontraremos tres categorías según el modelo en el que se relacionan los hilos a nivel de usuario y los hilos a nivel de kernel:

- Modelo 1:1
- Modelo N:1
- Modelo 1:N

- Modelo N:M

3.1.1 Hilos a nivel de usuario (ULT).

En una aplicación ULT pura, todo el trabajo de gestión de hilos lo realiza la aplicación y el núcleo o kernel no es consciente de la existencia de hilos. Es posible programar una aplicación como multihilo mediante una biblioteca de hilos. La misma contiene el código para crear y destruir hilos, intercambiar mensajes y datos entre hilos, para planificar la ejecución de hilos y para salvar y restaurar el contexto de los hilos.



Modelo de Usuario

Todas las operaciones descritas se llevan a cabo en el espacio de usuario de un mismo proceso. El kernel continua planificando el proceso como una unidad y asignándole un único estado (Listo, bloqueado, etc.) Las ventajas de los ULT son:

- El intercambio de los hilos no necesita los privilegios del modo kernel, por que todas las estructuras de datos están en el espacio de direcciones de usuario de un mismo proceso. Por lo tanto, el proceso no debe cambiar a modo kernel para gestionar hilos. Se evita la sobrecarga de cambio de modo y con esto el sobrecoste u overhead.
- Se puede realizar una planificación específica. Dependiendo de que aplicación sea, se puede decidir por una u otra planificación según sus ventajas.

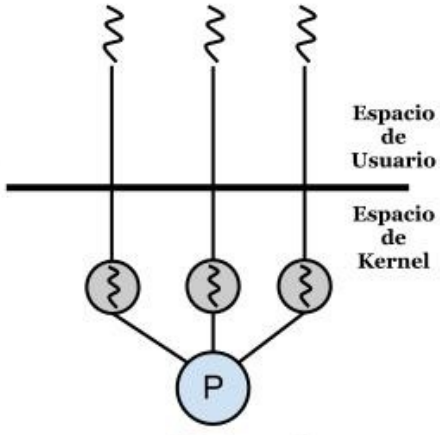
- Los ULT pueden ejecutar en cualquier sistema operativo. La biblioteca de hilos es un conjunto compartido.

  Igualmente las desventajas más relevantes son:

- En la mayoría de los sistemas operativos las llamadas al sistema son bloqueantes. Cuando un hilo realiza una llamada al sistema, se bloquea el mismo y también el resto de los hilos del proceso.
- En una estrategia ULT pura, una aplicación multihilo no puede aprovechar las ventajas de los multiprocesadores. El núcleo asigna un solo proceso a un solo procesador, ya que como el núcleo no interviene y ve al conjunto de hilos como un solo proceso.

3.1.2 Hilos a nivel de kernel (KLT).

En una aplicación KLT pura, todo el trabajo de gestión de hilos lo realiza el kernel. En el área de la aplicación no hay código de gestión de hilos, únicamente un API (interfaz de programas de aplicación) para la gestión de hilos en el núcleo.



Modelo de Kernel

Las principales ventajas de los KLT:

- El kernel puede planificar simultáneamente múltiples hilos del mismo proceso en múltiples procesadores.
- Si se bloquea un hilo, puede planificar otro del mismo proceso.

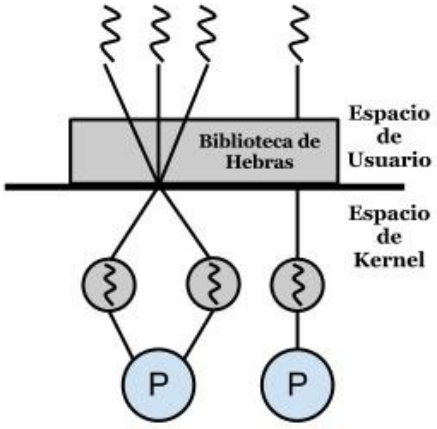
- Las propias funciones del kernel pueden ser multihilo

  En cambio las desventajas que presentan son:

- El paso de control de un hilo a otro precisa de un cambio de modo, lo que implica numeroso cambios de contexto.

3.1.3 Combinaciones ULT-KLT.

Algunos sistemas operativos ofrecen la combinación de ULT y KLT, como Solaris. La creación de hilos, así como la mayor parte de la planificación y sincronización de los hilos de una aplicación se realiza por completo en el espacio de usuario. Los múltiples ULT de una sola aplicación se asocian con varios KLT. El programador puede ajustar el número de KLT para cada aplicación y máquina para obtener el mejor resultado global.



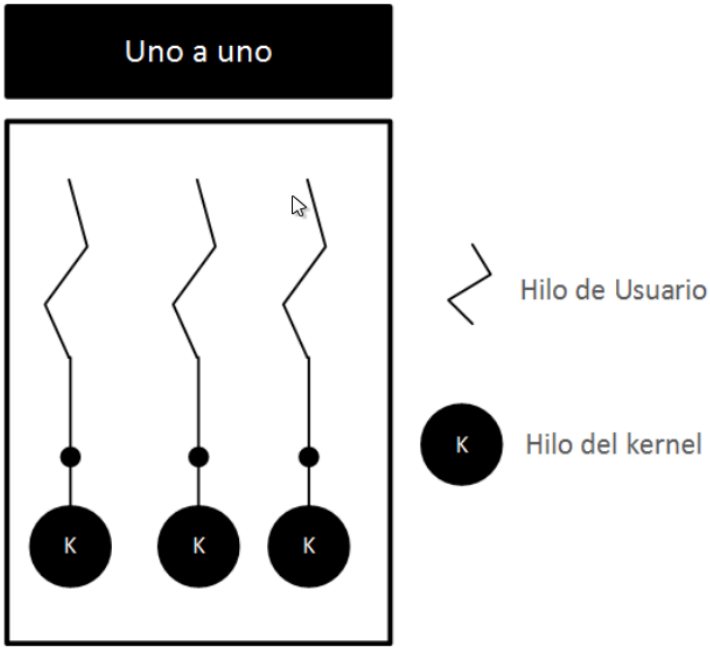
Modelo Híbrido

Si bien no hay muchos ejemplos de sistemas de este tipo si se pueden encontrar implementaciones híbridas auxiliares en algunos sistemas como por ejemplo las planificaciones de activaciones de hilos que emplea la librería NetBSD native POSIX threads.

3.2.1 Modelo 1:1

En este tipo de modelo se asigna un hilo de usuario a un hilo de núcleo. Es decir, cada hilo de ejecución es un único proceso con su propio espacio de

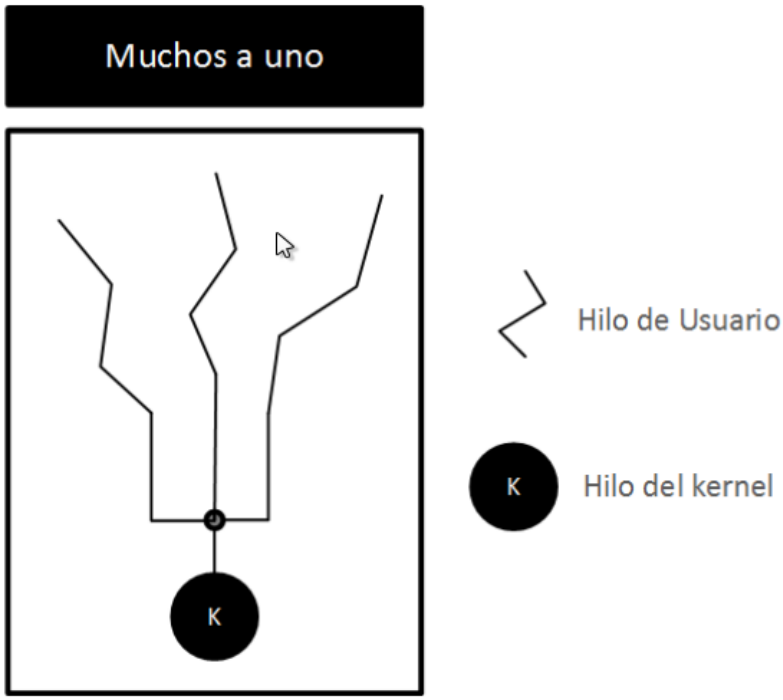
direcciones y recursos. Mediante este modelo la concurrencia es mayor, ya que si un hilo realiza una llamada bloqueante los demás hilos siguen ejecutándose. De esta forma se permite la ejecución de múltiples hilos en paralelo sobre varios procesadores. La desventaja del uso de este modelo radica en que la creación de cada hilo de usuario necesita la correspondiente creación de un hilo de núcleo. La implementación de este modelo se ve limitado en el número de hilos soportados en el sistema, debido a la carga que puede significar la creación de estos en la eficiencia del mismo. Algunos ejemplos de sistemas operativos que implementan este tipo de modelo son las implementaciones UNIX tradicionales, y windows.



3.2.2 Modelo N:1

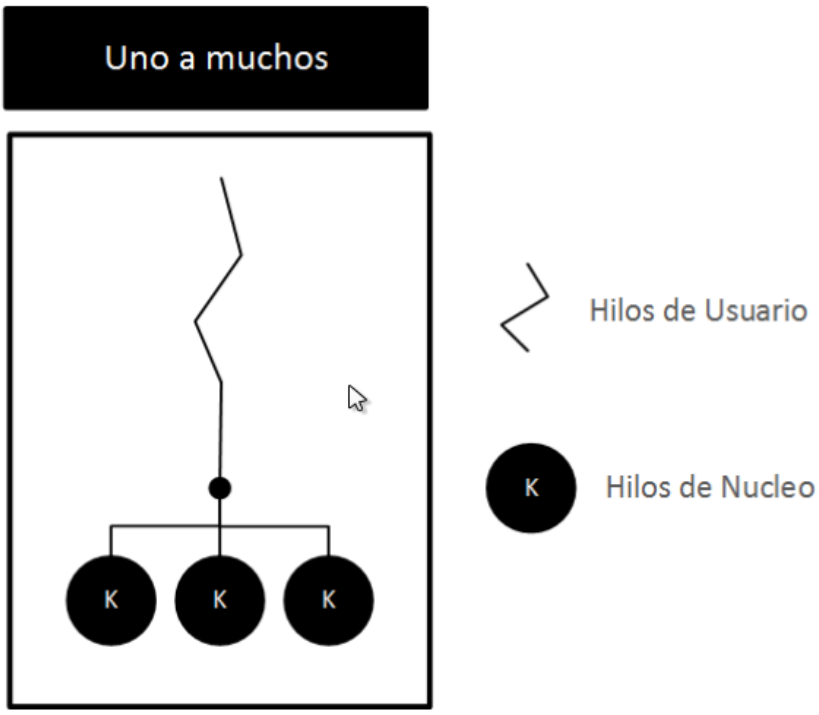
Este enfoque de modelo asigna múltiples hilos de nivel de usuario a un hilo de nivel de núcleo. La administración concerniente de los hilos se lleva a cabo mediante la biblioteca de hilos en el espacio de usuario. El espacio de direcciones, así como la pertenencia dinámica de recursos es definida por un proceso, de modo tal que se pueden crear y ejecutar varios hilos en este. No obstante, solo un hilo a la vez puede acceder al núcleo, de forma tal que no se pueden ejecutar paralelamente varios hilos. El proceso completo se bloquea si un hilo que

pertenece a este realiza una llamada bloqueante al sistema. Algunos de los sistemas operativos que hacen uso de este tipo de en modelo son Windows NT, OS/390, Solaris, Linux, OS/2 y MACH.



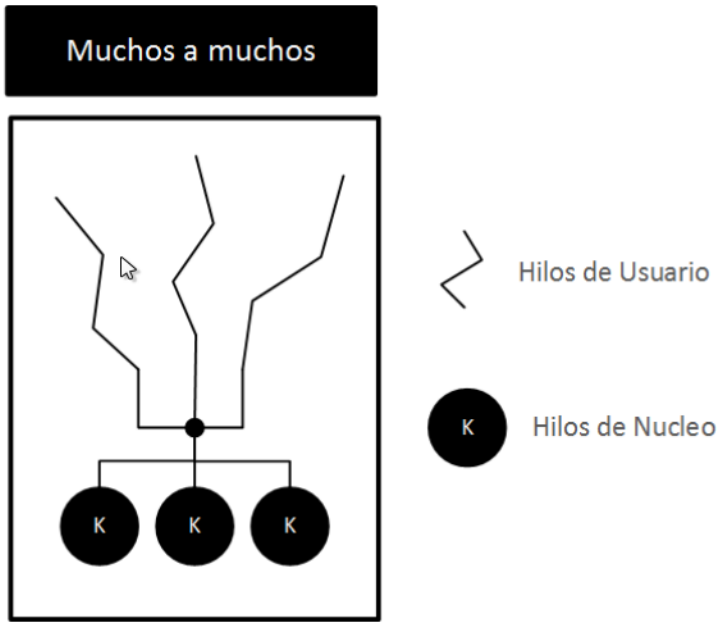
3.2.3 Modelo 1:N

La migración de un entorno de proceso a otro se lleva a cabo mediante este tipo de modelo, permitiendo con ello a los hilos moverse fácilmente entre los diferentes sistemas. Este tipo de entorno es de interés en los sistemas operativos distribuidos por la visualización de hilo como una entidad que puede movilizarse entre diferentes espacios de direcciones. Desde el punto de vista de usuario el hilo es una unidad de actividad y el proceso es un espacio de direcciones virtuales con el bloque de control de proceso asociado debidamente. Los hilos pueden movilizarse de un espacio de direcciones a otro, incluso de un computador a otro, para ello debe mantener consigo información tal como el control de terminal, parámetros globales y las guías de planificación. Algunos de los sistemas operativos que utilizan este tipo de modelo son las llamadas “nubes”.



3.2.4 Modelo N:M

Combinación de los modelos de Muchos a Uno y Uno a Muchos. Con este enfoque se multiplexan varios hilos de nivel de usuario de sobre un número de hilos, el cual es menor o igual a la cantidad de hilos de kernel. La cantidad de hilos de núcleo puede ser específica de un equipo o aplicación determinada. Este modelo permite la creación de hilos, siendo la cantidad de esta ilimitada ya que va de acuerdo a las necesidades existentes. Si se produce una llamada bloqueante al sistema por parte de un hilo, el núcleo puede planificar otro hilo para su ejecución. Con esto es posible ejecutar una actividad de un usuario o aplicación en múltiples dominios. Un ejemplo de sistema operativo que utiliza este enfoque es TRIX.



4. Estándares para el uso de hilos.

En este apartado abordaremos las distintas formas de en las que se emplean hilos en las distintas plataformas y API.

4.1 Hilos Posix

Posix es una librería con la definición de las funciones del manejo de los hilos para UNIX, como se vio anteriormente POSIX significa Portable Operating System Interface, donde la X es de UNIX. La librería POSIX contiene los estándares de manejo de hilos, esta API esta para C/C++, la "Carnegie MellonUniversity" dice: "Es más eficaz en sistemas de varios procesadores o varios núcleos donde el flujo de proceso puede ser programado para funcionar en otro procesador así ganando velocidad a través de procesamiento distribuido o paralelo". Está claro que los hilos de ejecución que aprovechan la velocidad de procesamiento de los diferentes núcleo de los procesadores aumente de manera considera el rendimiento del programa, un programa puede ser una operación aritmética, donde cada parte de la operación se resuelva con hilo de ejecución, y cada hilo asignado a un núcleo del procesador distinto, esto, claramente, puede lograr un mayor rendimiento, aunque el proceso de programación ser un poco

tedioso los resultados pueden ser sorprendentes.

Una de las ventajas de la programación en multiprocesadores, es la latencia o la espera, esta se de cuando un hilo se tiene que ejecutar mientras otro hilo se ejecuta y este espera una entrada o salida. La latencia no solo se da a nivel de hilos sino también a nivel de entrada y salida. Este tipo de espera es común en los procesadores de un solo núcleo, de ahí que sea una gran diferencia con los procesadores multinúcleo ya que la entrada o salida puede venir de otro núcleo del procesadores reduciendo la latencia a 0.

Posix nos aporta una gran variedad de funciones para el manejo de hebras, a continuación citaremos algunas:

- pthread_create: Esta es la función para la creación de los hilos (Pthreads).
- pthread_detach: Cambia el estado de un hila a detached (separado).
- pthread_equal: Comparar los identificadores de 2 hilos.
- pthread_exit: Termina la llamada a un hilo.
- pthread_getspecific: Gestiona los datos de un hilo en especifico.
- pthread_join: Espera la terminación de otro hilo.
- pthread_key_create: Gestiona los datos de un hilo en especifico.
- pthread_kill_other_threads_np: Termina todos los hilos en ejecución excepto la llamada al hilo.
- pthread_kill(pthread_sigmask): Maneja las señales a los hilos.
- Étc.

4.2 Hilos Java

Los hilos de ejecución en java provienen de la clase Thread, para la creación de los hilos en java es necesario crear un nuevo objeto de tipo Thread. La clase Thread viene en la librería de la Maquina Virtual de Java (JVM). Cuando se crean hilos en java estos pueden tener un alto o bajo nivel de

prioridad, los hilos de alta prioridad se ejecutan con preferencia sobre los de baja prioridad. La prioridad de los hilos es definida cuando se crean, y esta es heredada de su hilo creador, y es un hilo daemon si y sólo si el hilo creador es un daemon. Los hilos tipo daemon o hilos de utilidad son hilos que proveen servicios a otros hilos. La vida de los hilos tipo daemon dependen de la de los hilos de usuarios, cuando mueren todos los hilos de usuarios los hilos daemon tambien mueren, esta es la definición de segun JavaTpoint.

Al inicio de la maquina virtual de java, se crea un único hilo nodaemon, que típicamente llama el método main de alguna clase. La máquina virtual de java solo detiene los hilos de ejecución cuando ocurre alguna de las siguientes acciones:

- Cuando el método exit es llama de la clase Runtime y el manejador de seguridad permite que la operación exit pueda ser ejecutada.
- Cuando todos los hilos de tipo no-daemon mueren, por una devolución de la llama del método de ejecución o por el lanzamiento de un excepción que vaya mas allá del método de ejecución.

De igual forma existen 2 métodos para la creación de hilos en Java:

- Declarando una clase como subclase de la clase Thread. Esta subclase debe anular el método run de la clase Thread. Se ubicauna instancia de la subclase para iniciar un hilo.
- La otra forma es declarando una clase que implemente la interfaz Runnable. Posteriormente dicha clase implementará el método run. Se instanciará la clase asignando los argumentos cuando se cree el hilo y se indica.

4.3 Hilos Windows

Los hilos de Windows utilizan la API Win32, esta API de Windows utiliza la función de CreateThread para la creación de hilos, esta función recibe una serie de parámetro como atributos de seguridad, tamaño, rutina de inicio, entre otros. Una vez creado el hilo la función devolverá el manejador del nuevo hilo, en caso

de error la devolución de la función CreateThread seria NULL.

Entre las limitaciones presentes en los hilos de Windows, esta la cantidad de hilo que puede crear un proceso, ya que estos estarán limitados por la cantidad de memoria virtual que se encuentre disponible. La cantidad pre-establecida es de un megabyte de espacio en la pila. Los que nos muestra que la cantidad maxima de hilos son 2048 por defecto,esta cantidad puede aumentar, si se reduce la cantidad de espacio de pila para cada hilo. Una recomendación para el buen funcionamiento de una aplicación es crear un hilo de ejecución por proceso y también crear una cola de solicitudes, de esta manera se puede mantener el contexto de la información.

La de controlador de un hilo se puede arreglar con el derecho de THREADALL_ACCESS_, este llama un descriptor de seguridad este se crear para el nuevo hilo, utilizando un token primario para le proceso. Cuando se llama el OpenThreat el token evalúa los derechos, para darle o no permiso de acceso al hilos de ejecución.

Para terminar los hilos en windows se utiliza la función ExitThread, esta función utiliza el parámetro de lpStartAddress que se define al crear el hilos, para solicitar este ato se utiliza la función GetExitCodeThread, la cual nos proporciona la DWORD, que se utilizar como parámetro de ExitThread. Cuando el hilos es terminado este cambia su estado a Señalado, el cual avisa a los otros hilos en espera por el objeto.

5. Gestión de hilos por el programador.

En este último punto analizaremos desde una mayor abstracción la gestión que ha de hacer el programador sobre los hilos de ejecución sobre un programa. La gestión se realiza sobre librerías o API que ayudan a los programadores.

5.1. Semáforos

Un semáforo es una *estructura* u objeto que nos va a permitir la sincronización multiproceso o multihilo de nuestro programa. Básicamente se basa en un contador, una función de incrementar el contador y otra de decrementar el

contador. El control de acceso se realiza mediante el valor de ese contador, si el contador es negativo el hilo o proceso se suspenderá hasta que ese contador deje de ser negativo, momento en el cual el hilo o proceso empezará a ejecutarse automáticamente.

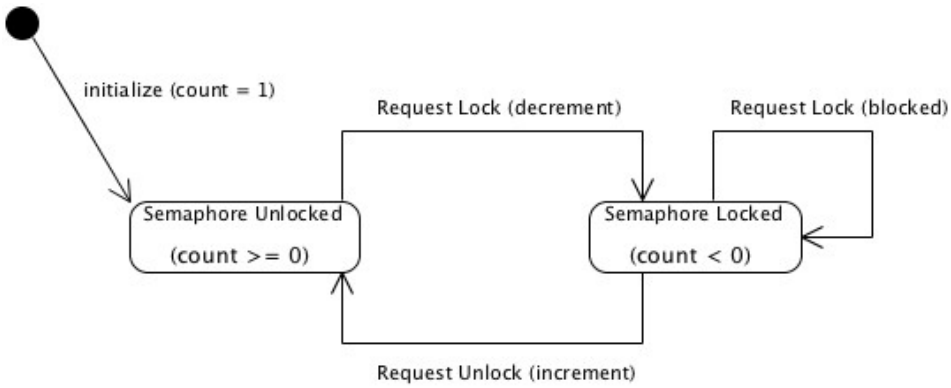
La implementación de los semáforos varía entre plataformas, por ejemplo en Linux están implementadas a nivel del sistema operativo dentro de los IPCs, POSIX ofrece un estándar de más alto nivel que usaremos para solucionar el problema anterior.

5.1.1 Funciones de los semáforos.

Además de la inicialización que veremos más delante, en la implementación de semáforos ante un problema concreto, hay dos funciones básica para gstionar un semáforo que son:

- sem_wait(s):
  - Si el valor asociado a s es mayor que cero, decrementa dicho valor en 1.
  - En otro caso (si el valor asociado a s es cero), bloquea el proceso que invoca en el conjunto de procesos bloqueados a s.
- sem_signal(s):
  - Si el conjunto de procesos asociados a s no está vacío, desbloquear uno de dichos procesos.
  - En otro caso (si el conjunto de procesos bloqueados a s está vacío, incrementa en una unidad el valor de s.

Los semáforos de un instante cualquiera de tiempo se ha de cumplir que el valor inicial asociado al semáforo más la cantidad de sem_signals realizados ha de ser igual que la suma del valor actual del semáforo más la cantidad de sem_wait realizados (no se incluyen los sem_wait no completados, es decir, lo que no ha dejado el proceso en espera). Hay que destacar que las operaciones sem_signal y sem_wait son atómicas, es decir, nunca habrá dos procesos distintos ejecutando estas operaciones a la vez sobre un mismo semáforo



5.1.3 Sección crítica.

La sección critica en programación concurrente, es la porción de código de un programa de ordenador en la que se accede a un recurso compartido (estructura de datos o dispositivo) que no debe ser accedido por más de un proceso o hilo en ejecución. La sección crítica por lo general termina en un tiempo determinado y el hilo, proceso o tarea sólo tendrá que esperar un período determinado de tiempo para entrar. Se necesita un mecanismo de sincronización en la entrada y salida de la sección crítica para asegurar la utilización en exclusiva del recurso, por ejemplo un semáforo. El acceso concurrente se controla teniendo cuidado de las variables que se modifican dentro y fuera de la sección crítica. La sección crítica se utiliza por lo general cuando un programa multihilo actualiza múltiples variables sin un hilo de ejecución separado que lleve los cambios conflictivos a esos datos. Una situación similar, la sección crítica puede ser utilizada para asegurarse de que un recurso compartido, por ejemplo, una impresora, pueda ser accedida por un solo proceso a la vez.

5.2 Monitores.

En un sistema pueden existir algunos procesos que necesitan comunicarse entre sí, compitiendo además por los recursos del sistema, una zona de datos específicos o un dispositivo de E/S, y dicha competición se debe regular a través de la sincronización de los procesos. Las técnicas de sincronización son el fundamento del procesamiento concurrente. Existen dos métodos básicos de

comunicación entre procesos: “Compartición de datos” e “Intercambio de información”. En este tema se trata inicialmente el problema de la comunicación entre procesos mediante “Compartición de datos”. Los “Semáforos” constituyen el mecanismo básico fundamental para el problema de la sincronización pero dado que en la utilización de éstos pueden existir errores de temporización debido a su incorrecto uso y a que son difíciles de detectar, es que nace la idea de “Monitores” que proporcionan un mecanismo alternativo al de los Semáforos.

5.2.1 Desventajas frente a los semáforos.

Algunas de las ventajas de los monitores frente al uso de semáforos son las siguientes:

- El uso y la función de las variables no es explícito.
- Están basados en variables globales → No es modular.
- Las operaciones sobre las variables dispersas y no protegidas.
- Acceso no estructurado ni encapsulación → Fuente de errores.

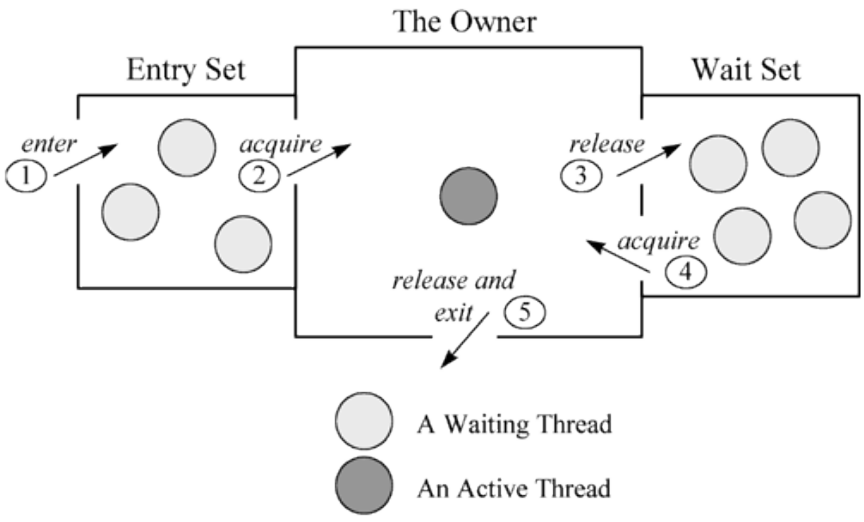
5.2.2 Componentes de un monitor.

Un monitor tiene cuatro componentes: inicialización, datos privados, métodos del monitor y cola de entrada.

- Inicialización: contiene el código a ser ejecutado cuando el monitor es creado
- Datos privados: contiene los procedimientos privados, que sólo pueden ser usados desde dentro del monitor y no son visibles desde fuera
- Métodos del monitor: son los procedimientos que pueden ser llamados desde fuera del monitor.
- Cola de entrada: contiene a los hilos que han llamado a algún método del monitor pero no han podido adquirir permiso para ejecutarlos aún.

5.2.3. Exclusión mutua en un monitor.

Los monitores están pensados para ser usados en entornos multiproceso o multihilo, y por lo tanto muchos procesos o threads pueden llamar a la vez a un procedimiento del monitor. Los monitores garantizan que en cualquier momento, a lo sumo un thread puede estar ejecutando dentro de un monitor. Ejecutar dentro de un monitor significa que sólo un thread estará en estado de ejecución mientras dura la llamada a un procedimiento del monitor. El problema de que dos threads ejecuten un mismo procedimiento dentro del monitor es que se pueden dar condiciones de carrera, perjudicando el resultado de los cálculos. Para evitar esto y garantizar la integridad de los datos privados, el monitor hace cumplir la exclusión mutua implícitamente, de modo que sólo un procedimiento esté siendo ejecutado a la vez. De esta forma, si un thread llama a un procedimiento mientras otro thread está dentro del monitor, se bloqueará y esperará en la cola de entrada hasta que el monitor quede nuevamente libre. Aunque se la llama cola de entrada, no debería suponerse ninguna política de encolado.



Para que resulten útiles en un entorno de concurrencia, los monitores deben incluir algún tipo de forma de sincronización. Por ejemplo, supóngase un thread que está dentro del monitor y necesita que se cumpla una condición para poder continuar la ejecución. En ese caso, se debe contar con un mecanismo de

bloqueo del thread, a la vez que se debe liberar el monitor para ser usado por otro hilo. Más tarde, cuando la condición permita al thread bloqueado continuar ejecutando, debe poder ingresar en el monitor en el mismo lugar donde fue suspendido. Para esto los monitores poseen variables de condición que son accesibles sólo desde adentro. Existen dos funciones para operar con las variables de condición:

- cond_wait(c): suspende la ejecución del proceso que la llama con la condición c. El monitor se convierte en el dueño del lock y queda disponible para que otro proceso pueda entrar.
- cond_signal(c): reanuda la ejecución de algún proceso suspendido con cond_wait bajo la misma condición c. Si hay varios procesos con esas características elige uno. Si no hay ninguno, no hace nada.

Nótese que, al contrario que los semáforos, la llamada a cond_signal(c) se pierde si no hay tareas esperando en la variable de condición c.

Las variables de condición indican eventos, y no poseen ningún valor. Si un thread tiene que esperar que ocurra un evento, se dice espera por (o en) la variable de condición correspondiente. Si otro thread provoca un evento, simplemente utiliza la función cond_signal con esa condición como parámetro. De este modo, cada variable de condición tiene una cola asociada para los threads que están esperando que ocurra el evento correspondiente. Las colas se ubican en el sector de datos privados visto anteriormente.

La política de inserción de procesos en las colas de las variables condición es la FIFO, ya que asegura que ningún proceso caiga en la espera indefinida, cosa que sí ocurre con la política LIFO (puede que los procesos de la base de la pila nunca sean despertados) o con una política en la que se desbloquea a un proceso aleatorio.

5.3. Problema del productor consumidor.

El problema del productor-consumidor es un ejemplo clásico de problema de sincronización de multiprocesos. El programa describe dos procesos, productor

y consumidor, ambos comparten un buffer de tamaño finito. La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) productos uno a uno. El problema consiste en que el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío.

La idea para la solución es la siguiente, ambos productos se ejecutan simultáneamente y se "despiertan" o "duermen" según el estado del buffer. Concretamente, el productor agrega productos mientras quede espacio y en el momento en que se llene el buffer se pone a "dormir". Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente. En caso contrario si el buffer se vacía, el consumidor se pone a dormir y en el momento en que el productor agrega un producto crea una señal para despertarlo.

Se puede encontrar una solución usando mecanismos de comunicación interprocesos, generalmente se usan semáforos. Una inadecuada implementación del problema puede terminar en un deadlock donde ambos procesos queden en espera de ser despertados. Este problema pude ser generalizado para múltiples consumidores y productores.

5.3.1 Solución P-C con semáforos.

La solución propuesta con semáforos que veremos es la aplicada en las practicas de otra asignatura, en SCD, Sistemas Concurrentes y Distribuidos. La explicación del código la relazaremos dentro de los comentarios del mismo.

```
#include <iostream>        // Para la entrada y salida por pantalla.
#include <cassert>         // Control de excepciones
#include <pthread.h>       // Para el manejo de hebras
#include <semaphore.h>     // Uso de las funciones de los semáforos
#include <unistd.h>        // Necesario para {\ttbf usleep()}
#include <stdlib.h>        // Necesario para {\ttbf random()}, {\ttbf srandom()}
#include <time.h>          // Necesario para {\ttbf time()}

using namespace std ;

////////////////////////////////////////////////////////////////////////////////
// Constantes configurables:
```

```
//
const unsigned
   num_items  = 40 ,    // numero total de items que se producen o consumen
   tam_vector = 10 ;    // tamaño del vector, debe ser menor que el número de items

////////////////////////////////////////////////////////////////////////////
//   Semaforos:
//   Como podemos observar los semáforos son variables de tipo sem_t, proveniente de la
//   biblioteca <semaphore.h>

sem_t   puedo_leer,        //  Semaforo para las hebras lectoras
        puedo_escribir,    //  Semáforo para las hebras escritoras
        mutex ;            //  Semáforo para garantizar la exclusión mútua.

////////////////////////////////////////////////////////////////////////////
// Buffer y control del tamaño, implementación LIFO
//

int    v[tam_vector],
       primera_libre = 0;

////////////////////////////////////////////////////////////////////////////
//  Función que introduce un retraso aleatorio de duración comprendida entre
// 'smin' y 'smax' (dados en segundos) para comprobar que nuestra implementación
// "funciona" correctamente. Simulamos que una lectura y escritura van a tener
// tiempos distintos.
//
void retraso_aleatorio( const float smin, const float smax ){
   static bool primera = true ;
   if ( primera ){                 // Si es la primera vez que llamamos entonces...
      srand(time(NULL));      //  Inicializar la semilla del generador
      primera = false ;           //  No repetir la inicialización
   }
   // calcular un número de segundos aleatorio, entre {smin} y {smax}
   const float tsec = smin+(smax-smin)*((float)random()/(float)RAND_MAX);
   // dormir la hebra (los segundos se pasan a microsegundos, multiplicándos por 1 millón)
   usleep( (useconds_t) (tsec*1000000.0));
}

////////////////////////////////////////////////////////////////////////////
// Función que simula la producción de un dato.
//
unsigned producir_dato(){
   static int contador = 0 ;          // Dato producido
   contador = contador + 1 ;          //
   retraso_aleatorio( 0.1, 0.5 );      // Retraso antes de mostrar la producción
```

```
    cout << "Productor : dato producido: " << contador << endl << flush ;
    return contador ;
}

//////////////////////////////////////////////////////////////////////////
// Función que simula la consumición de un dato.
//
void consumir_dato( int dato ) {
   retraso_aleatorio( 0.1, 1.5 );
   cout << "Consumidor:   dato consumido: " << dato << endl << flush ;
}

//////////////////////////////////////////////////////////////////////////
// Función que ejecuta la hebra productora.
//
void * funcion_productor( void * ) {
    for( unsigned i = 0 ; i < num_items ; i++ ) {
        sem_wait(&puedo_escribir);      // Cada vez que produzco un dato he de consumir uno
        int dato = producir_dato() ;    // una unidad del valor sociado al semaforo escritor.

        sem_wait( &mutex);              // El semáforo mutex que segura la exclusión mutua
        v[primera_libre] = dato;        // Introduzco el dato
        primera_libre++;                // Modifico cual es la siguiente pos, libre
        cout << "Productor : dato insertado: " << dato << endl << flush ;
        sem_post(&mutex);               // Finalizo la zona de exclusión mutua.

        sem_post(&puedo_leer);          // Envio la seña de que ya puedo leer al semaforo asociado
    }                                   // En cada ciclo se incrementa el valor del semáforo.
    return NULL ;
}

//////////////////////////////////////////////////////////////////////////
// Función que ejecuta la hebra del consumidor
//
void * funcion_consumidor( void * ) {
    for(unsigned i = 0 ; i < num_items; i++ ) {
        int dato ;
        sem_wait(&puedo_leer);          // Si el valor de puedo_leer es 0 (no hay datos producidos)
                                        // el proceso se bloquea, en caso contrario se consume 1.
        sem_wait(&mutex);               // Sección crítica
        primera_libre--;                // Se reduce en uno ya que se extraerá un dato.
        dato = v[primera_libre];        // Se guarda en una variable intermedia.
        cout << "Consumidor:       dato extraído : " << dato << endl << flush ;
        consumir_dato( dato ) ;         // Se consume el dato.
        sem_post(&mutex);               // Notifico la salida de la región crítica.
```

```
        sem_post(&puedo_escribir);      // Se incrementa el valor de puedo_escribir para
    }                                   // que así se pueda producir otro dato.
    return NULL ;
}

//////////////////////////////////////////////////////////////////////////////////////

int main(){

    pthread_t hebra_escritora, hebra_lectora ; // Declaración de las hebras.

    // sem_init inicializa el semáforo, el último parámetro indica el valor con el que se inicializa
    // puedo_escribir se inicializa a tam_vector para que así se puedan producir como máximo
    // hasta el número asociado al tam del vector.
    sem_init( &mutex,              0,  1 );
    sem_init( &puedo_leer,         0,  0 );
    sem_init( &puedo_escribir,  0,  tam_vector ); // Puedo escribir tan


    // Crea la hebras y además se le asocia una función que será la que ejecute.
    pthread_create( &hebra_escritora, NULL, funcion_productor,  NULL) ;
    pthread_create( &hebra_lectora,   NULL, funcion_consumidor, NULL) ;

    // Función que espera a la terminación de ejecución de las hebras
    pthread_join(hebra_escritora, NULL);
    pthread_join(hebra_lectora,   NULL);

    // Destrucción de los semáforos tras la finalización de las hebras.
    sem_destroy( &puedo_escribir );
    sem_destroy( &puedo_leer );
    sem_destroy( &mutex );

  return 0 ;
}
```

5.3.2 Solución P-C con monitores.

Al igual que la solución al problema del Productor-Consumidor con semáforos, esta solución es la aplicada en las practicas de Sistemas Concurrentes y Distribuidos con monitores de tipo Hoare.

```
import monitor.*;    // Paquete para el uso de monitores Hoare
                     // Link de descarga:
                     // https://www.dropbox.com/s/awfew3mhuyivgha/monitor.zip?dl=0

// ************************************************************************** //

/**
 * Clase Buffer.
 * Clase que implementa un buffer de dobles con funciones de
 * extracción e inserción de datos que funciona como monitor para
 * los procesos que extraigan e inserten en el buffer.
 * @autor Antonio Miguel Morillo Chica
 */
class Buffer extends AbstractMonitor {
    // Colas de condición, los procesos bloqueados esperan en la cola según cuando
    // hagamos los signal() o los await() sobre la cola.
    private Condition puedo_poner = makeCondition();
    private Condition puedo_quitar = makeCondition();
    private int tam = 0;          // Tam de memoria que reservamos.
    private int ocupados = 0;   // Posición hasta donde está ocupado.
    private double[] buffer = null;  // Declaración del buffer

    /**
     * Constructor por parámentros de la clase
     * @param tam tamaño a reservar que necesitamos
     */
    public Buffer(int tamBuffer){
        tam = tamBuffer;
        buffer = new double [tam];
    }

    /**
     * Función que añade un valor al buffer.
     * @param valor que se quiere introducir
     *
     * Funcionalmente lo que hacemos es esperar cuando el buffer está lleno.
     * Sino introducimos el valor, incrementamos los ocupados variable que sabemos
     * que es compartida por las hebras pero gracias a enter() garantizamos que se hará
     * en exclusión mutua, con leave() saldremos de ella.
     */
    public void depositar(double valor) throws InterruptedException{
        enter();                        // Incicio de la EM

        if (tam == ocupados)          // Si el vector está lleno → esperar
            puedo_poner.await();
                                        // En caso contrario continuo la ejecución
```

```
        buffer[ocupados] = valor;      // Introduzco el valor
        ocupados++;                    // Incremento la cantidad de ocupados
        puedo_quitar.signal();         // En este momento puedo liberar uno de los procesos que
                                       // esperan en la cola de puedo_quitar.
        leave();                       // Fin de la EM
    }

    /**
     * Funcion que elimina un valor del buffer
     * @return valor extraidos.
     *
     * Funcionalmente sigue la misma filosofía que poner un dato, la única diferencia
     * que no puedo quitar nada si el vector está vacío por ello se realiza la
     * comprobación de si los ocupados=0.
     */
    public double extraer() throws InterruptedException{
        enter();
        double valor;
        if (ocupados == 0)
            puedo_quitar.await();
        ocupados--;
        valor = buffer[ocupados];
        puedo_poner.signal();
        leave();

        return valor;
    }
}

/**
 * Clase que simula la produción de un dato.
 * @autor Antonio Miguel Morillo Chica
 */
class Productor implements Runnable{
    private Buffer   buf                ;
    private int      numeroProducciones ;
    private int      idHebra            ;
    public  Thread   thr                ;

    /**
     * Constructor por parámetros.
     * @param   newBuf buffer donde guardará datos.
     * @param   nProd numero de producciones que se harán
     * @param   id identificador de la hebra Runnable
     * @return  devuelve el objeto implicito
     */
```

```
    public Productor( Buffer newBuf, int nProd, int id ){
        buf                 =   newBuf;
        numeroProducciones  =   nProd;
        idHebra             =   id;
        thr                 =   new Thread(this,"Hebra Productora  [" +idHebra+ "]");
    }

    /**
     * El método run es el corazón del subproceso, es donde tiene lugar la acción del
     * subproceso. Hay dos modos de proporcionar el el método run a un subproceso:
     */
    public void run(){
        try{
            double item = 100*idHebra ;

            for( int i=0 ; i < numeroProducciones ; i++ ){
                System.out.println(thr.getName()+" -> produce [" + item + "]");
                buf.depositar( item++ );
            }
        }
        catch( Exception e ){
            System.err.println("Excepcion en main: " + e);
        }
    }
}

/**
 * Clase que simula a un consumidor.
 * No se especifica el nombre de la hebra que simulará a un consumidor pues
 * lo que se hará será preguntar a la propia clase que hebra está actuando como
 * un consumidor.
 * @autor Antonio Miguel Morillo Chica
 */
class Consumidor implements Runnable{
    private Buffer   buf             ;
    private int      nConsumiciones  ;
    private int      idHebra         ;
    public  Thread   thr             ;

    /**
     * Constructor por defecto de la clase.
     * @param   newBuf buffer que se usará para realizar lecturas.
     * @param   nCons numero de lecturas a consumir
     * @param   id identificador de la hebra
     * @return  devuelve el objeto implicito.
     */
```

```
    public Consumidor( Buffer newBuf, int nCons, int id ){
        buf             =  newBuf;
        nConsumiciones  =  nCons;
        idHebra         =  id ;
        thr             =  new Thread(this,"Hebra Consumidora ["+idHebra+"]");
    }

    /**
     * El método run es el corazón del subproceso, es donde tiene lugar la acción del
     * subproceso. Hay dos modos de proporcionar el el método run a un subproceso:
     */
    public void run(){
        try{
            for( int i=0 ; i<nConsumiciones ; i++ ){
                double item = buf.extraer ();
                System.out.println(thr.getName()+" -> consume ["+item+"]");
            }
        }
        catch( Exception e ){
            System.err.println("Excepcion en main: " + e);
        }
    }
}

/**
 * Clase que simula la comunicación entre productores y consumidores.
 * @autor Antonio Miguel Morillo Chica
 */
class MainProductorConsumidor {
    public static void main(String[] args) {
        if (args.length != 5) {
            System.err.println("USO: <nProductores> <nConsumidores> <tamBuffer>
                                    <numProducciones> <numConsumiciones>");
        return;
    }


    Productor[]  productoras   = new Productor[Integer.parseInt(args[0])];
    Consumidor[] consumidoras  = new Consumidor[Integer.parseInt(args[1])];
    Buffer     buffer          = new Buffer(Integer.parseInt(args[2]));
    int        iProducciones   = Integer.parseInt(args[3]);
    int        iConsumiciones  = Integer.parseInt(args[4]);

    if (productoras.length*iProducciones != consumidoras.length*iConsumiciones ){
      System.err.println("No coinciden número de items a producir con a cosumir\n");
      return ;
```

```
    }

    // Inicializo las hebras consumidoras a la vez que las creo.
    for (int i = 0 ; i<consumidoras.length; i++) {
      consumidoras[i] = new Consumidor(buffer, iConsumiciones, i);
    }
    // Inicializo las hebras productoras a la vez que las creo
    for (int i = 0 ; i<productoras.length; i++) {
      productoras[i] = new Productor(buffer, iProducciones, i);
    }
    // Inicializo las hebras productoras a la vez que las creo
    for (int i = 0 ; i<productoras.length; i++) {
      productoras[i].thr.start();
    }
    // Inicializo las hebras consumidoras a la vez que las creo.
    for (int i = 0 ; i<consumidoras.length; i++) {
      consumidoras[i].thr.start();
    }
  }
}
```

FIN