**Informe Proyecto (BETA)**

**CONJUNTO DE TAREAS**

Para el ejercicio “Conjunto de tareas” sincronizado con **hilos y semáforos** se decidió por tener un arreglo de 6 semáforos (uno por tarea para saber cuándo finaliza un ciclo) y un semáforo extra para la exclusión mutua y que solo una tarea se realice por vez; y la cantidad de hilos utilizados varía de acuerdo a la cantidad de tareas que se deben realizar, sean 4, 5 o 6.

Para compilar el archivo: gcc <nombreArchivo.c> -o <nombreSalida> -pthread

realizarTarea() { //válido sea para la tarea A, B y C

wait(ciclado);

ejecutarTarea();

signal(semTarea);

signal(ciclado);

En el caso de la sincronización con **pipes**, el padre se comunicará con cada tarea que debe realizar mediante un pipe. Se optó por declarar un pipe, y el proceso padre lo creará, realizará la tarea requerida y recibirá por dicho pipe el resultado obtenido.

Se optó por tener una estructura para almacenar el mensaje que el proceso hijo le envía al padre para hacer más legible el código.

En cuanto al código, los procedimientos *realizarTareaB(int tipo)* y *realizarTareaC(int tipo)* podrían refactorizarse y unificarse en un solo método (en ambos solo cambia el mensaje que se envía, y en el caso de la tarea C se suma la generación de llantas para reparar). Por una cuestión de legibilidad y no otorgar la mayoría del control a un solo procedimiento, se decidió dejarlos separados.

Para compilar el archivo: gcc <nombreArchivo.c> -o <nombreSalida>

cicloTareas() {

fork()

si es el proceso hijo

realizarTarea()

si es el proceso padre

read(pd)

print(mensaje)

}

realizarTarea() { //válido para tarea A, B y C

realizarTarea()

write(pd)

}

**MINI SHELL**

La idea principal de este programa, es simular el funcionamiento de una consola.

Cuando el usuario ingresa una instrucción, el proceso se encarga de separar la cadena de caracteres ingresada en varias subcadenas. Para separar el comando ingresado y sus argumentos se utilizó la función ***\*strtok(char \*str, const char \*delim).*** Luego se discrimina la operación a realizar, dentro de las indicadas en el enunciado. Finalmente se ejecuta o se controlan errores.

Los comandos a utilizar son:

* *mkdir*: Crear un directorio con el nombre ingresado por parámetro.
* *rmdir*: Eliminar directorio con el nombre pasado por parámetro.
* *touch*: Crear archivo con el nombre pasado por parámetro.
* *ls*: Mostrar una lista de archivos de un directorio pasado por parámetro. Se utiliza la librería dirent.h. Funciona únicamente si los nombres de las carpetas de la dirección pasada por parámetro no contienen algún espacio
* *cat*: Mostrar el contenido de un archivo pasado por parámetro.
* *chmod*: Modificar los permisos de un archivo.

**DEMASIADAS BOTELLAS DE LECHE**

Para el ejercicio “Demasiadas botellas de leche” sincronizado con **hilos** **y semáforos** se optó por un semáforo binario para sincronizar a los compañeros (*comp*) y uno (también binario) para la leche (*comprarLeche*). Inicialmente el semáforo que sincroniza los compañeros se encuentra habilitado, indicando que está habilitado para que uno abra el refrigerador y chequee si hay leche. Como solo puede haber una leche almacenada, al abrir como no habrá, utilizando el semáforo “comprarLeche” se le permitirá solo a uno ir a comprar leche. Es en este punto que se ingresa en la sección crítica, para lo cual es usado dicho semáforo.

Para compilar el archivo: gcc <nombreArchivo.c> -o <nombreSalida> -pthread

Para ejecutar: ./<nombreSalida>

*Pseudocódigo:*

abastecerLeche() {

wait(comp);

if(try wait (comprarLeche) == 0) { //sección crítica

ir al super y comprar leche;

guardar leche

}

else {

signal(comprarLeche);

}

signal(comp);

}

En el caso de la sincronización con **colas de mensajes** se creó una cola para sincronizar los N compañeros que pueden eventualmente llegar a la heladera a tomar leche y una cola para enviar a un compañero a comprar leche. Inicialmente una cola tendrá N mensajes (uno por cada compañero) y otra cola no tendrá mensajes, y el tarro de leche estará habilitado para ser consumido.

El problema que sucede en este ejercicio es que se atenderá de a un solo mensaje de la cola que quiere tomar leche, por lo que si al llegar un mensaje no hay leche, la cola con dichos mensajes se bloqueará hasta que un mensaje atienda la falta de leche y la reponga.

Para compilar cada archivo: *gcc <nombreArchivo.c> -o <nombreSalida> -lrt*

Para ejecutar:

los archivos refrigerador.c y compras.c: ./<nombreSalida>

el archivo principal.c: ./<nombreSalida><cantidad de compañeros>

*Pseudocódigo:*

refrigerador()

si hay mensajes en la cola de compañeros que quieren ver el refrigerador

recibir un mensaje de un compañero que quiere tomar leche

si hay que comprar leche

enviar mensaje de tipo comprar leche a la cola de comprar leche

recibir mensaje de tipo comprar leche de la cola ver refrigerador

guardar la leche

no hay que comprar leche

sino

tomar leche

hay que comprar leche

enviar mensaje con el compañero a la cola de ver refrigerador

compras()

si hay un mensaje en la cola de comprar leche

recibir mensaje de la cola de comprar leche

comprar leche

enviar mensaje a la cola de ver el refrigerador de tipo comprar leche

principal()

crear N mensajes en la cola de compañeros que van a ir al refrigerador

**COMIDA RÁPIDA**

Para el ejercicio “comida rápida” sincronizado con **hilos** se optó por tener un hilo para cada cocinero (3 en total), uno para el camarero, un hilo para el limpiador y un hilo por cada cliente (50 en total). En lo que respecta a los semáforos, se tiene uno de tipo contador para las mesas que están disponibles para ser usadas; otro semáforo contador para las mesas que quedan sucias al retirarse el cliente, un semáforo contador para indicar si hay un cocinero libre, un semáforo para exclusión mutua en el caso que se solicite al camarero, uno contador para la comida que se debe cocinar, un semáforo binario para indicar que el limpiador puede habilitar una mesa sucia para ser usada.

En cuanto a la solución encontrada sincronizando con hilos, se debe iniciar los hilos clientes después de haber iniciado los camareros, cocineros y limpiadores. Si se inicializan primero, entrarán 30 clientes para ocupar las mesas sucias y el sistema quedará interbloqueado porque no se pueden atender los pedidos por no tener un camarero que sirva la comida ni cocineros que ingresen comida a la cola de comidas. En el caso del limpiador, también debe inicializarse antes de los clientes porque si las 30 mesas son usadas no podrán ingresar nuevos clientes si no hay mesas limpias.

En el caso de la sincronización con **colas de mensajes** se optó por una cola con los clientes que esperan para entrar al restaurante, otra cola para aquellos que están para ser atendidos, una cola para los pedidos que realizan los clientes, los cuales se modelaron de manera tal que, actualmente con la pandemia,los restaurantes habilitaron el pedido mediante un código QR sin involucrar un camarero que tome el pedido, realizandolo directamente desde un dispositivo y solo el camarero entrega la comida. Así, en la solución propuesta el cliente directamente hace un pedido que tomará el cocinero mediante una cola de pedidos, lo cual facilitará que en la cola de comida lista que cocinarán los cocineros no se llene de un solo tipo de comida y solo se prepare la que se necesita. Para los cocineros se optó por tenerlos en una cola, al igual que las mesas limpias, inicialmente todas en una cola y una cola vacía para las mesas sucias que se irá llenando cuando el cliente que la usó se retire.

Es importante en este caso respetar el orden de ejecución de los archivos para que no se produzcan inconsistencias en la salida, como así también interbloqueos. En primer lugar, se debe ejecutar el archivo inicio.c; luego, indistintamente de su orden, los archivos camarero.c, limpiador.c y cocineros.c. Por último el archivo cliente.c.

*Pseudocódigo:*

limpiador()

recibir mensaje a la cola mesa sucia

enviar mensaje a la cola mesa limpia

camarero()

si hay clientes para atender

recibir un mensaje de la cola atender cliente

si msj == c //carne

recibir mensaje tipo 1 de cola comida lista

si msj == v //vege

recibir mensaje tipo 2 de comida lista

entregar plato al cliente

cliente()

si hay clientes en la cola para atender

si hay mesas disponibles en la cola de mesas limpias

recibir un cliente de la cola de clientes para atender

recibir un mensaje de la cola de mesas limpias

el cliente pide el menú que eligió

enviar el pedido para que un cocinero lo prepare

enviar un mensaje a la cola para ser atendido por el camarero

comer

irse

enviar un mensaje a la cola de mesas sucias

cocinero()

si hay algún cocinero ocioso

recibir un cocinero de la cola

si hay pedidos que cocinar

recibir un mensaje de la cola de pedidos

si hay lugar en la cola de la comida lista para entregar

cocinar la comida que se pidió (carne o vege)

enviar un mensaje a la cola de comida lista para entregar

**Problemas**

Artículo elegido: “Android Operating System Architecture”

**a) Describa los componentes de la arquitectura.**

La arquitectura de este sistema operativo se basa en cuatro capas:

* Kernel de Linux
* Librerías nativas, conteniendo la librería Android y Android Runtime
* Framework de aplicaciones
* Aplicaciones

La capa del **kernel de Linux** provee la funcionalidad estándar del sistema. Este kernel fue modificado por Google para atender las necesidades que un sistema móvil requiere. Así, se contará con el driver de alarma para manejar de manera correcta ciertos eventos. También tendrá los controladores Binder para manejar la comunicación entre procesos debido a que un dispositivo móvil tiene los recursos más limitados que una PC. La modificación del kernel de Linux incorpora el binder permite que se solucionen ciertos problemas de costos de procesamiento y de seguridad. El kernel también tendrá a su cargo la gestión de energía, sumamente importante para celulares, tablets y todo dispositivo que tenga este sistema operativo y para funcionar no dependa de estar conectado a la corriente, por lo que tiene que estar a la altura de las circunstancias y manejar de la mejor manera posible los recursos. En lo que respecta a los controladores de memoria, se debe hacer una eficiente gestión de los procesos, por lo que si no hay memoria suficiente, existe un manejador de memoria que finaliza de manera inteligente aquellos procesos que corren en segundo plano. Cuando un dispositivo presenta poca memoria, la memoria compartida permite tener un mejor soporte y su gestión será más eficiente. Otra parte importante de este kernel es su debugger, que permite mantener de manera correcta el funcionamiento de Android en caso que un tercero haga cambios en el sistema.

Luego está la capa de **librerías**, también llamada capa nativa, que incluye un conjunto de librerías en C/C++ usadas en varios componentes de Android. Están compiladas en código nativo del procesador. Muchas de las librerías utilizan proyectos de código abierto. En lo que respecta a las librerías, está libc, WebKit, SQLite, el framework de media (audio, video y fotografía), el administrador de superficies, librerías 3D y SSL. Libc proporciona funcionalidad básica para la ejecución de las aplicaciones; es una versión adaptada de BSD, pero adaptada para sistemas embebidos de Linux. Luego está WebKit, el motor web utilizado por el navegador, que permite visualizar correctamente contenido HTML. En lo que respecta a SQLite, esta base de datos relacional es usada en Android para almacenar datos y se encuentra disponible para todas las aplicaciones que la requieran. El framework de media soporta codecs de reproducción y grabación de multitud de formatos de audio y video e imágenes. El administrador de superficies maneja el acceso al subsistema de representación gráfica en 2D y 3D.

La capa de librerías contiene una parte dedicada al entorno de ejecución. El componente principal del entorno de ejecución de Android es la máquina virtual Dalvik, componente que ejecuta todas y cada una de las aplicaciones no nativas de Android. Las aplicaciones se compilan en un formato específico para la máquina virtual Dalvik, que es la que las ejecuta.

Sobre la capa de librerías se encuentra la capa de **framework de aplicaciones**. Esta la forman las clases y servicios que usan directamente las aplicaciones para realizar sus funciones. La mayoría de los componentes son bibliotecas Java. Así, la componen el administrador de actividades, de ventanas, el proveedor de contenidos, el administrador de paquetes, el de recursos, las vistas, administrador de telefonía, servicios de localización, entre otros. El administrador de actividades va a controlar el ciclo de vida de las mismas; estas actividades son como las “ventanas” que se muestran en una aplicación. Para mostrar en la pantalla se requerirá del administrador de ventanas, que generará superficies que pueden ser rellenadas por las actividades. El proveedor de contenidos por otro lado permite encapsular los datos que serán compartidos entre aplicaciones, como SQLite y el sistema de archivos por ejemplo. El manejador de paquetes permite obtener información de los paquetes que actualmente se encuentran instalados en el dispositivo. El administrador de telefonía proporciona el acceso a los servicios de telefonía del dispositivo; permite realizar llamadas y enviar y recibir SMS. El administrador de ubicaciones permite determinar la posición geográfica del dispositivo.

Por último, la capa superior está formada por las **aplicaciones**. Se incluye a las aplicaciones del dispositivo, tanto las pre instaladas como aquellas que el usuario instala.

**b) Identifique elementos que son representativos para este tipo de sistemas operativos**

En el caso de este sistema operativo que actualmente lidera el mercado en móviles, tiene un un kernel de Linux que está modificado por Google. Esta modificación se da debido a que dicho kernel debe ser adaptado a los usos que puede llegar a tener un móvil. Android tiene sus propias librerías como son SQLite y Bionic (anteriormente libc) para vincular y compilar. Vale también aclarar que tiene su propio entorno de programación, el runtime con librerías y las máquinas virtuales para poder programar. El contar con estas herramientas permite que un desarrollador tenga a su alcance lo necesario para producir software de calidad, adaptado a las necesidades del usuario.

**c) Realice un comentario general del artículo.**

Según lo visto en las clases de la materia, Android es un sistema operativo híbrido al igual que los sistemas operativos actuales porque la estructura de los mismos no es puramente de un tipo.

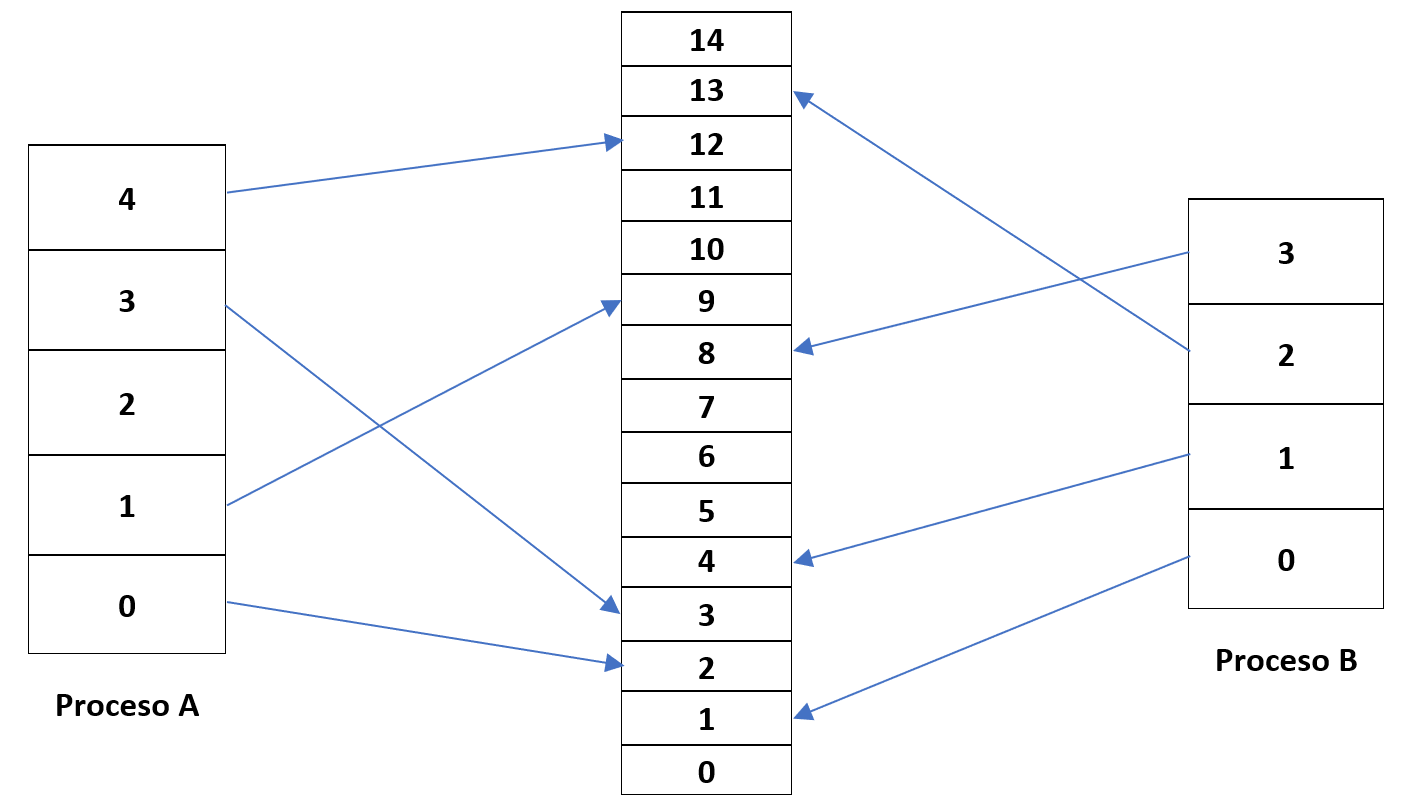
En la última década hubo un avance exponencial en materia de tecnología, que ha ayudado en el día a día de las personas. Tal es así que el uso de dispositivos móviles ha superado en gran cantidad a las computadoras personales en parte porque su utilidad es similar. Con un teléfono inteligente no solo se realiza llamadas o se envía y recibe mensajes de texto como en aquellos viejos dispositivos de la década de 1990 y 2000 sino que llegan a ser una extensión del humano; permite tener (casi) todo en la palma de la mano. Sea un dispositivo con Android o iOS, es posible realizar tareas, trámites, como también entretenerse.

En primer lugar, se puede decir que la arquitectura estructurada en capas en Android permite que cada capa resuelva un único conjunto de problemas, y el usuario final sólo interactuará con la capa “más superficial”, es decir la capa de aplicaciones, centrada en la ejecución, comunicación y estabilidad de toda aplicación preinstalada por cada fabricante como de las aplicaciones que el usuario va a instalar. Esta capa permite tener un gran nivel de comprensión y simplicidad para el usuario. Por otro lado, un desarrollador construirá las aplicaciones que se encontrarán en dicha capa utilizando las herramientas y abstracciones que provee la capa de framework de aplicaciones, la cual simplifica el acceso a componentes de bajo nivel al crear API’s sobre librerías nativas. En la capa de librerías nativas y de entorno de ejecución se utiliza lenguajes de bajo nivel; es la capa que optimiza los dispositivos móviles. Esto garantiza que el código que desarrolla un programador pueda ejecutarse sin problemas a pesar de las limitaciones del dispositivo. En la parte inferior se encuentra la capa perteneciente al kernel de Linux, que interactúa con los componentes de hardware comunes en los dispositivos Android.

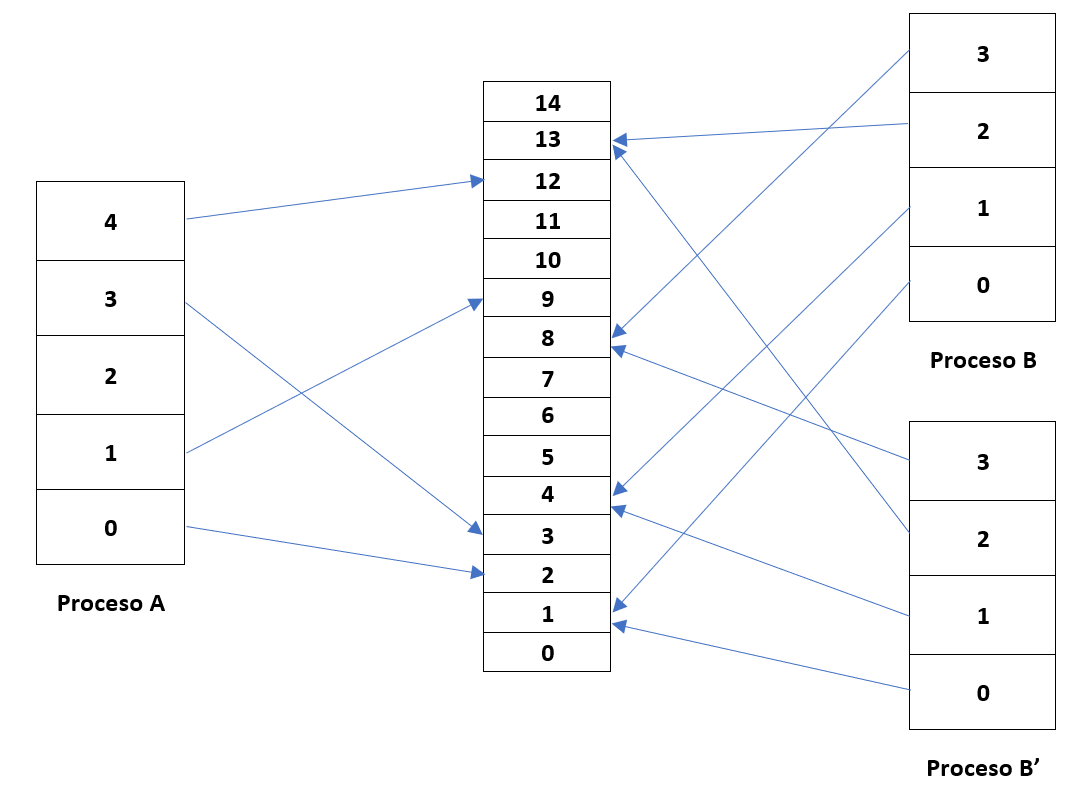
Este sistema de código abierto está presente en un abanico de marcas, las cuales deben adaptarlos a sus dispositivos. Las actualizaciones de seguridad están aseguradas por un tiempo limitado para cada dispositivo, por lo que se puede sufrir de ataques a los que el sistema puede llegar a ser muy vulnerable. En lo que respecta a la seguridad, cabe destacar que toda aplicación requiere de permisos para cada funcionalidad que posea: otorgar permisos que la aplicación no requiera puede ser motivo para preocuparse por la integridad del dispositivo involucrado.

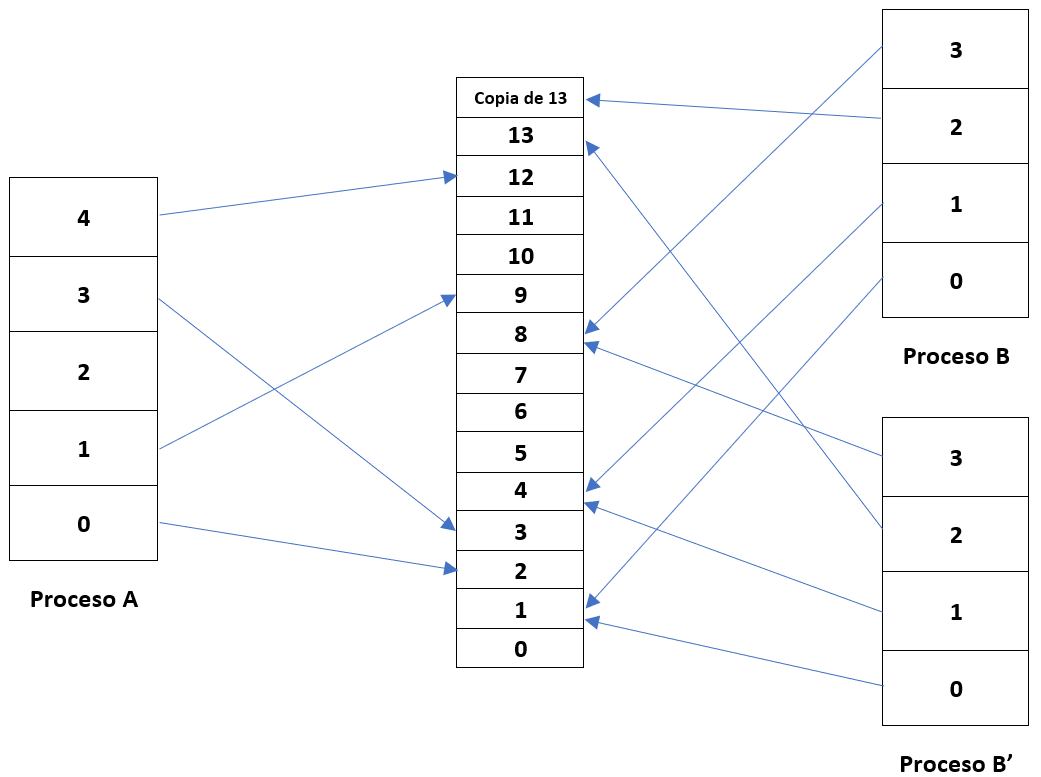
**Problemas Conceptuales**

**1- La figura 1 muestra la asignación de páginas en un sistema Unix que ejecuta los procesos A y B. Las páginas son de 4 KB.**

****

**a) Suponga que el proceso B invoca fork y que el kernel utiliza la estrategia copy-on-write para implementar fork. Realice un nuevo diagrama para la asignación de páginas después de invocar fork y luego de que el proceso B modificó la página 2.**

* **Después de invocar fork()**
* **Después que el proceso B modificó la página 2**



El proceso B modifica la página 2 de su espacio de direcciones, por lo tanto mapea a la dirección física 13 en memoria. Como el proceso B modifica la página 13, duplica la misma.

**b) Construya la tabla de páginas para el padre y el hijo indicando los atributos de cada página (bits de válido y escritura).**

* **Tabla de páginas del proceso padre (Proceso B)**

**Proceso B:**

|  |  |  |
| --- | --- | --- |
| **Página** | **Bit de válido** | **Escritura** |
| **3** | **1** | **0** |
| **2** | **1** | **1** |
| **1** | **1** | **0** |
| **0** | **1** | **0** |

**Proceso B’:**

|  |  |  |
| --- | --- | --- |
| **Página** | **Bit de válido** | **Escritura** |
| **3** | **1** | **0** |
| **2** | **1** | **0** |
| **1** | **1** | **0** |
| **0** | **1** | **0** |