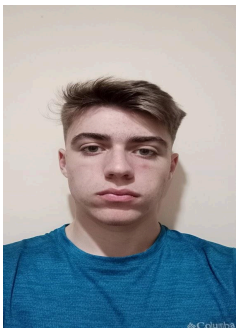
Obligatorio Sistemas Operativos

Nicolás Bidenti (305108)

Santiago Canadell (282542)

Felipe Delgado(281987)





ÍNDICE:

Ejercicio 1)

El menú principal es el núcleo del programa. Muestra tres opciones principales: registrar un administrador, ingresar al sistema o salir. Según la elección, redirige lo que se seleccionó con un bloque case,hicimos funciones específicas para manejar cada acción.
La función registrar_usuario permite ingresar nuevos usuarios al sistema. Pide los datos como nombre, cédula, teléfono, fecha de nacimiento y tipo de usuario. Verificamos que la cédula no esté ya registrada utilizando grep y, dependiendo del tipo de usuario, guarda los datos en Usuarios_clientes.txt o Usuarios_admin.txt.
La función registrar_mascotas hace el registro de nuevas mascotas. Se asegura de que cada mascota tenga un identificador único usando las validaciones con grep. Guarda información como tipo, nombre, sexo, edad y descripción. Los datos se guardan en Datos_mascotas.txt junto con la fecha de ingreso y el estado inicial de disponibilidad.
La función estadisticas_adopcion hace los reportes sobre el sistema. Usamos while para recorrer los datos de Datos_mascotas.txt, agrupando estadísticas por tipo de mascota y mes de adopción. También calculamos la edad promedio de los animales adoptados.
La función listar_mascotas muestra todas las mascotas disponibles para adopción.Con un while filtramos mascotas "disponible", leyendo y mostrando los datos en Datos_mascotas.txt.
La función adoptar_mascota pide el identificador de la mascota que se quiere adoptar, verifica si está disponible y actualiza su estado en Datos_mascotas.txt utilizando sed. También registra la adopción en adopciones.txt junto con la fecha actual, usando el comando date para obtenerla.
Los menús redirigen dinámicamente a las funciones según las entradas del usuario, y los datos los procesamos con comandos como read, grep, y sed para búsquedas, validaciones y actualizaciones de manera eficiente.

Ejercicio 2)

Representación de las Materias
Las materias las representamos con un struct Materia que tiene:

- codigo: Identificador de la materia (por ejemplo, "IP" para Introducción a la Programación).
- predecesores: Lista de punteros a las materias de las que depende.
- numPredecesores: Número de predecesores que deben completarse antes de ejecutar la materia.
- cond y mutex: Variables para sincronizar la ejecución mediante exclusión mutua y condiciones.
- ejecutada: Indicador de si la materia ya fue ejecutada.

Inicialización de Materias y Dependencias
En la función inicializarMaterias creamos las instancias de las materias, se asignan sus predecesores según el grafo de dependencias y configuramos las variables de sincronización. Las dependencias entre materias se definen usando un array global de materias.
Ejecución en Paralelo con Threads
imprimirMateria se ejecuta en hilos independientes para procesar cada materia. Antes de ejecutarse, la materia espera a que todas sus predecesoras se completen, usando las variables de condición y mutex. Esto para estar seguro que las dependencias sean respetadas.

- Esperar predecesores: El hilo comprueba el estado de ejecución de cada materia predecesora y espera con pthread_cond_wait si alguna no está completa.
- Ejecución de la materia: Una vez que las dependencias se completan, imprime el código de la materia y marca su estado como ejecutado.
- Notificar a las dependientes: Señala a las materias dependientes que pueden continuar su ejecución con pthread_cond_signal.

Flujo Principal
En la función main:

1. Se inicializan las materias y con las dependencias (inicializarMaterias).
2. Se crean los hilos para ejecutar cada materia usando pthread_create.
3. Esperamos a que todos los hilos terminen su ejecución con pthread_join.
4. Al final, se libera la memoria asignada para las estructuras y variables.

Ejercicio 3)

Usamos constantes para representar las instrucciones como LOAD, STORE, ADD, SUB, entre otras. También , implementamos un tipo protegido Semaphore para manejar las operaciones de sincronización Wait, Signal e Init.
Declaramos un tipo protegido Shared_Memory que hace la lectura y escritura segura en la memoria compartida.
Implementación de Semáforos
El tipo Semaphore hace la sincronización mediante operaciones (Wait) y (Signal). La entrada Init inicializa el valor del semáforo. Estas operaciones aseguran exclusión mutua y coordinación entre CPUs.
El tipo protegido Shared_Memory permite leer y escribir en la memoria compartida de manera segura. Hicimos la memoria como un array de 128 posiciones, con verificaciones de rango para evitar acceder a una posición inválida.
Cada CPU se define como una tarea (Task) que ejecuta instrucciones desde un programa cargado en memoria. Cada CPU tiene su propio acumulador (Accumulator) y puntero de instrucción (IP). Las instrucciones se procesan en un bucle, e incluyen:
LOAD, STORE, ADD y SUB son instrucciones fundamentales que manipulan el acumulador y la memoria compartida. LOAD extrae un valor desde una dirección de memoria y lo carga en el acumulador utilizando Shared_Memory_Task.Read.
STORE guarda el valor del acumulador en una posición específica de la memoria mediante Shared_Memory_Task.Write.
Las instrucciones ADD y SUB realizan operaciones aritméticas sumando o restando un valor leído de la memoria al acumulador, utilizando la misma operación de lectura antes de actualizar el acumulador.
SEMWAIT, SEMSIGNAL y SEMINIT son instrucciones para la sincronización mediante semáforos. SEMWAIT bloquea la CPU si el semáforo no está disponible hasta que se libere con Semaphores(Semaphore_ID).Wait, mientras que SEMSIGNAL incrementa el semáforo, permitiendo que otras tareas continúen su ejecución con Semaphores(Semaphore_ID).Signal. SEMINIT inicializa un semáforo con un valor dado, configurándolo con Semaphores(Semaphore_ID).Init para coordinar las operaciones concurrentes.
BRCPU es una instrucción condicional que ajusta el puntero de instrucción (IP) dependiendo del identificador de CPU. Si la CPU actual no coincide con el identificador esperado, la ejecución salta a una dirección específica, asegurando que solo el procesador correspondiente continúe con las instrucciones.
PRINT y STOP son instrucciones de salida y finalización. PRINT lee un valor de la memoria compartida y lo imprime en la consola mediante Shared_Memory_Task.Read y Put_Line.
STOP terminala ejecución de la CPU, parando su bucle principal.

Flujo Principal del Programa
En el bloque principal:

1. Inicializamos los semáforos y memoria compartida: Los semáforos aseguran sincronización entre CPUs.
2. Cargamos las instrucciones en memoria: Se configuran los programas para ambas CPUs.
3. Creamos las tareas CPU: Cada CPU inicia su ejecución de manera independiente.

4. Sincronizamos la ejecución: Los semáforos garantizan que las instrucciones se ejecuten en el orden correcto.

Consideración de semáforos:
Utilizamos dos semáforos en el programa, uno que controla el acceso a la memoria, inicializado en 1 para solamente permitir el acceso de una CPU; y otro que controla la finalización de ambas CPU para luego realizar la impresión del resultado, este lo inicializamos en -1 ya que cada vez que una CPU termina su ejecución le suma 1 al contador del semáforo, permitiendo que cuando ambos terminan la CPU 1 acceda al semáforo e imprima el resultado.

Consideración de inicio de las CPU:
Iniciamos cada CPU en una posición de memoria determinada para asegurarnos que nunca acceden a posiciones de memoria de la otra, si bien se podría haber realizado con BRCPU decidimos realizarlo en la inicialización por simple practicidad.

Ejemplo de Ejecución
Inicializamos una variable llamada valor en 8. La CPU 0 suma 13 a esta variable, mientras que la CPU 1 suma 27. Finalmente, el resultado se imprime utilizando la instrucción PRINT. Este flujo asegura consistencia mediante la sincronización de semáforos.

Ejercicio 4)

Documentación del Ejercicio: Levantar SQL Server con Docker

Paso 1: Levantar SQL Server utilizando Docker
Usamos el comando docker run para descargar e iniciar un contenedor con SQL Server. Configuramos las variables de entorno necesarias como ACCEPT_EULA=Y para aceptar los términos de uso y SA_PASSWORD=nico123! como la contraseña para el usuario administrador. También asignamos el puerto 1433 para la conexión y nombramos el contenedor como sql server. Esto permitió levantar una instancia de SQL Server en su versión más reciente disponible en Docker Hub.

Paso 2: Verificar que el contenedor esté en ejecución
Verificamos que el contenedor estuviera funcionando correctamente utilizando el comando docker ps. Este comando nos mostró una lista de los contenedores activos, confirmando que el contenedor de SQL Server estaba en ejecución y listo para recibir conexiones en el puerto 1433.

Paso 3: Configurar la conexión con DBeaver
Una vez que el contenedor estuvo en funcionamiento, configuramos una nueva base de datos en DBeaver para conectarnos al SQL Server. Proporcionamos los siguientes parámetros:
Host: localhost
Puerto: 1433
Nombre de usuario: sa
Contraseña: nico123!
Con esta configuración, logramos conectarnos correctamente y pudimos interactuar con la base de datos desde DBeaver.

Paso 4: Crear un archivo docker-compose.yaml
Para simplificar la gestión del contenedor, creamos un archivo docker-compose.yaml. En este archivo, definimos el servicio sqlserver, especificamos la imagen a utilizar, configuramos las variables de entorno necesarias, mapeamos el puerto 1433 y agregamos un volumen para guardar los datos de SQL Server en el host. Esto nos aseguró que los datos no se perdieran si el contenedor se detenía.

Paso 5: Levantar el contenedor con Docker Compose
Finalmente, utilizamos el comando docker-compose up -d para iniciar el contenedor definido en el archivo docker-compose.yaml. Este comando no solo levantó el contenedor, sino que también configuró los volúmenes automáticamente.