TEC | Tecnológico de Costa Rica

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación

Proyecto #1

MultiDisplay Animator

Manfred Jones Sanabria - 2022176476

Miguel Sánchez Sánchez - 2019061555

Prof. Kevin Moraga Garcia

IC6600

Principios de Sistemas Operativos

Semestre 1

01 de junio de 2025

1. Introducción

El objetivo del proyecto es lograr hacer uso de hilos y *schedulers* para coordinar movimientos de figuras en un *canvas*, a la vez que se re-implementa la biblioteca `mypthreads`.

Para lograrlo, se implementarán las siguientes bases del proyecto:

- **Lenguaje descriptivo de la animación:** un archivo contendrá las instrucciones de lo que sucederá en el *canvas*, indicando los monitores, las figuras, los movimientos, el tiempo y demás información importante para lograr la animación.
- **Hilos para el manejo de las figuras:** cada figura se manejará por medio de un hilo, el cual leerá las instrucciones del lenguaje de animación, ejecutará el movimiento correspondiente, evitará colisiones y cumplirá los tiempos.
- **Control de colisiones y espacio:** para verificar que no ocurran colisiones en el *canvas*, cada hilo correspondiente a una figura deberá asegurarse de que el espacio al que se moverá no esté ocupado por otra. Si dicho espacio está ocupado, el hilo deberá esperar a que se desocupe para realizar el movimiento o desaparecerá "explotando", esto con el uso de semáforos y *mutex*.
- **Canvas y monitores:** el *canvas* será el área donde se mueven las figuras y este, mediante una indicación del lenguaje de la animación, deberá poder cambiar de tamaño y distribuirse en diversas secciones que llamaremos *monitores* que pueden estar en distintas computadoras.
- **Distribución y sockets:** los *sockets* se utilizarán para la comunicación y sincronización entre los diferentes monitores en el sistema distribuido. Las máquinas compartirán y podrán acceder a la información de posición de los objetos en el *canvas* para evitar colisiones. Cada monitor cuenta con su propio hilo, y se coordinan mediante los semáforos o *mutex*.
- **Schedulers:** los *schedulers* se usarán para priorizar los hilos del programa. Habrá tres tipos: *RoundRobin* (predeterminado), sorteo y tiempo real. El *scheduler* a usar deberá ser seleccionado al momento de iniciar la ejecución del programa; si no, se asignará el predeterminado.

2. Ambiente de desarrollo

2.1. Lenguaje de programación

El proyecto se desarrollará utilizando el lenguaje de programación C.

2.2. Sistema operativo: Linux

El desarrollo se llevará a cabo en un entorno Linux/GNU. Las herramientas y bibliotecas estándar de Unix son ampliamente compatibles con Linux, lo que facilita la implementación de servidores y la administración de procesos.

2.3. Biblioteca `mypthreads` (pthreads)

Se utilizará una reemplementación de la biblioteca `pthread` y sus diferentes para crear y gestionar hilos de ejecución.

2.4. GCC

El compilador estándar de C en Linux, utilizado para compilar el código fuente.

2.5. VS Code

Se utilizará para escribir y editar el código fuente y otros archivos.

3. Estructuras de datos usadas y funciones

3.1. Módulo `mypthreads`

El módulo `mypthread` implementa una librería de hilos cooperativos en espacio de usuario, utilizando la biblioteca `ucontext.h`. Se simula una API similar a `pthread`, con funciones para crear hilos, hacer yield, join, detach y manejo de mutex. El código se divide en dos archivos principales:

- `mypthread.h`: Contiene las definiciones y prototipos.
- `mypthread.c`: Contiene la implementación de la lógica de manejo de hilos y mutexes.

Definición de tipos y estructuras

- `my_thread_t`: Identificador entero para los hilos.
- `my_mutex_t`: Estructura simple con bandera de bloqueo y dueño.
- `thread_state`: Enum para los estados del hilo (READY, RUNNING, FINISHED, BLOCKED).
- `thread_control_block`: Representa el contexto y estado de cada hilo.

Funciones principales

`my_thread_create()` Inicializa el contexto del nuevo hilo, configura su stack, lo marca como READY y lo agrega al scheduler.

`my_thread_end()` Termina el hilo actual. Si hay un hilo en `join`, lo despierta. Pasa el control al siguiente hilo del scheduler.

`my_thread_yield()` Cede voluntariamente el control al siguiente hilo listo.

`my_thread_join()` Bloquea el hilo actual hasta que el hilo objetivo finalice.

`my_thread_detach()` Marca el hilo como detached, lo que evita que otros hilos lo puedan unir.

`my_mutex_init()`, `lock()`, `unlock()`, `trylock()`, `destroy()` Funciones para manejo básico de exclusión mutua cooperativa usando spinlock y `yield()` para evitar espera activa prolongada.

`my_thread_chsched()` Cambia el tipo de scheduler asociado a un hilo. Esto puede ser útil si se implementan múltiples políticas de planificación.

Stack y tabla de hilos

Cada hilo cuenta con su propio stack de tamaño fijo definido por `STACK_SIZE`. Todos los hilos se almacenan en una tabla global `thread_table[MAX_THREADS]`, donde se guarda su contexto, estado y otra información necesaria.

3.2. Módulo de `scheduler`

El módulo `scheduler` es responsable de administrar la selección del siguiente hilo a ejecutar. Soporta múltiples políticas de planificación que pueden ser asignadas a los hilos: `ROUND_ROBIN`, `LOTTERY`, y `REAL_TIME`.

Tipos de Schedulers

- **ROUND_ROBIN**: Se recorren los hilos de forma circular, otorgando turnos iguales a cada hilo con esta política.
- **LOTTERY**: Se asignan "boletos"de lotería virtuales a los hilos, y se elige uno al azar ponderado por dichos boletos.
- **REAL_TIME**: Siempre se selecciona el hilo `READY` con el menor identificador (`id`), priorizando respuestas rápidas.

Estructura `Scheduler`

```
typedef struct {
    SchedulerType type;
    my_thread_t *queue;
    int count;
} Scheduler;
```

Contiene el tipo de scheduler activo, una cola (no utilizada completamente en esta versión), y un contador de hilos registrados.

Funciones principales

- `void scheduler_init(SchedulerType type)`: Inicializa el planificador con el tipo especificado.
- `my_thread_t scheduler_next()`: Selecciona el próximo hilo a ejecutar según el tipo de planificador. Verifica el estado y el tipo de planificación de cada hilo.
- `void scheduler_run()`: Ejecuta en bucle el planificador hasta que no haya más hilos `READY`, usando `setcontext` para cambiar al hilo siguiente.

Algoritmo ROUND_ROBIN

Se utiliza un índice global `rr_index` para recorrer los hilos de forma cíclica. Si el hilo está `READY` y es del tipo `ROUND_ROBIN`, se selecciona.

Algoritmo LOTTERY

Cada hilo `READY` recibe una cantidad de boletos proporcional a su identificador. Se genera un número aleatorio y se selecciona el hilo correspondiente al rango donde cae dicho número.

Algoritmo REAL_TIME

Se busca el hilo `READY` con el identificador más bajo, simulando una política de prioridad fija simple.

3.3. Módulo `canvas`

El módulo `canvas` define el lienzo principal donde se colocan y actualizan las figuras animadas. También permite registrar múltiples monitores virtuales que observan distintas regiones del lienzo en distintos tiempos.

Estructura `Canvas`

```
#define MAX_FIGURES 100
#define MAX_MONITORS 10

typedef struct Canvas {
    int width, height;
    Figure *figures[MAX_FIGURES];
    int figure_count;
    struct Monitor *monitors[MAX_MONITORS];
    int monitor_count;
} Canvas;
```

- `width, height`: Dimensiones del lienzo.

- `figures`: Arreglo de punteros a figuras activas.
- `figure_count`: Cantidad de figuras activas.
- `monitors`: Arreglo de punteros a monitores registrados.
- `monitor_count`: Número de monitores.

Funciones Principales

- `void canvas_init(Canvas *canvas, int width, int height)`
  Inicializa un lienzo con dimensiones específicas, sin figuras ni monitores.
- `void canvas_add_figure(Canvas *canvas, Figure *fig)`
  Añade una figura al lienzo si no se ha alcanzado el máximo (`MAX_FIGURES`).
- `void canvas_add_monitor(Canvas *canvas, Monitor *mon)`
  Añade un monitor al lienzo si no se ha alcanzado el máximo (`MAX_MONITORS`).
- `void canvas_update(Canvas *canvas, int current_time)`
  Actualiza las posiciones de todas las figuras activas (según su intervalo de tiempo) usando `figure_move`.
- `void canvas_draw(Canvas *canvas, int time, int monitor_id)`
  Dibuja el estado del lienzo desde la perspectiva de un monitor específico. Imprime las figuras visibles dentro de los límites del monitor en el tiempo indicado.

Comportamiento

Durante la ejecución, el lienzo mantiene un conjunto de figuras que pueden tener distintos intervalos de actividad (definidos por `start_time` y `end_time`) y trayectorias de movimiento. Las actualizaciones se realizan por cada unidad de tiempo, y los monitores pueden observar secciones rectangulares del lienzo, imprimiendo el estado visible de las figuras.

Ejemplo de Flujo

1. Se inicializa un lienzo con cierto tamaño.
2. Se añaden figuras y monitores al lienzo.
3. Por cada unidad de tiempo:
   - Se llama a `canvas_update` para mover las figuras activas.
   - Se llama a `canvas_draw` por cada monitor para imprimir la vista parcial del lienzo.

3.4. Módulo `parser`

El módulo `parser` se encarga de interpretar un archivo de texto que describe una animación compuesta por múltiples objetos, y de convertir dicha información en estructuras utilizables por el sistema.

Funciones principales

- `Figure* object_to_figure(const Object *obj);`
  Convierte un objeto de tipo `Object` en una figura de tipo `Figure`. Esta función es útil para traducir la representación abstracta a una figura que pueda ser procesada por el canvas.

- `int parse_animation(const char* filename, Animation* anim);`
  Parsea el archivo de texto de entrada que contiene instrucciones sobre la animación. Llena la estructura `Animation` con la información de ancho, alto y objetos animados. Retorna la cantidad de objetos leídos.

- **parse_animation:** Abre el archivo de animación, interpreta comandos como `CANVAS`, `OBJECT`, `CHAR`, `START`, `END`, `POSITION`, `MOVE TO` y `BOUNCE`, y almacena cada objeto en la estructura `Animation`.

- **draw_canvas:** Dibuja en la consola el estado del canvas para un instante de tiempo determinado. Utiliza un arreglo bidimensional de caracteres para representar el lienzo y coloca los objetos activos en sus respectivas posiciones.

- **update_objects:** Actualiza la posición de los objetos en función del tipo de movimiento (lineal o rebote) y del instante de tiempo actual.

- **run_animation:** Ejecuta la animación completa desde el tiempo 0 hasta el máximo tiempo de finalización entre todos los objetos. Cada frame se dibuja cada 100 milisegundos.

Comandos soportados en el archivo de entrada

- `CANVAS width height`: Define el tamaño del lienzo.

- `OBJECT id`: Inicia la definición de un nuevo objeto.

- `CHAR c`: Carácter que representa al objeto.

- `START t` y `END t`: Definen el intervalo de visibilidad del objeto.

- `POSITION x y`: Posición inicial del objeto.

- `MOVE TO x y DURATION t`: Movimiento lineal hacia una posición en un tiempo determinado.

- `BOUNCE LIMITS x1 y1 x2 y2 SPEED dx dy`: Movimiento de rebote dentro de una región con velocidades definidas.

3.5. Módulo `object_to_figure`

Este módulo tiene como objetivo convertir un objeto de tipo `Object`, que contiene información sobre una animación abstracta, en una estructura `Figure` que puede ser utilizada por el motor de renderizado. La conversión toma en cuenta el tipo de movimiento (lineal, rebote o ninguno), así como el tiempo de inicio y fin, el símbolo de representación, y otras propiedades necesarias para la animación visual.

Funciones principales

- `Figure* object_to_figure(const Object *obj);`
  Convierte un puntero a un `Object` en un puntero a una estructura `Figure` con los valores adaptados según el tipo de movimiento especificado en el objeto. Esta función reserva memoria dinámicamente para la figura generada y debe ser liberada manualmente cuando ya no se necesite.

Implementación

La función `object_to_figure` realiza lo siguiente:

1. Inicializa una estructura `Figure` con valores por defecto.
2. Copia las coordenadas iniciales del objeto a la figura.
3. Dependiendo del tipo de movimiento definido en `obj->movement.type`, configura los parámetros de movimiento:
   - Si el movimiento es `MOVE_LINEAR`, se asignan los puntos inicial y final, además de la duración.
   - Si es `MOVE_BOUNCE`, se copian las deltas de rebote (`dx`, `dy`).
   - Si no hay movimiento, se asigna el tipo `FIG_MOVE_NONE`.
4. Se copian los límites del movimiento (usualmente usados en rebote) y el tiempo de vida de la figura.
5. Se asigna el carácter visual y la etiqueta (`label`) de la figura usando `strncpy`.
6. Se reserva memoria dinámica para un puntero `Figure*` y se copia el contenido.
7. Finalmente, se imprime en consola un mensaje con los datos principales de la figura generada.

3.6. Módulo `monitor`

Este módulo define y gestiona un `Monitor`, el cual representa una región del lienzo (`Canvas`) que puede ser accedida de manera segura en entornos concurrentes mediante el uso de mutex. Su función principal es permitir visualizar el estado actual de una sección específica del lienzo en un momento dado.

Estructuras

- `Monitor`
  Representa una región rectangular del lienzo asociada a un `Canvas` y protegida por un mutex. Contiene las siguientes propiedades:
  - `int x0, y0, x1, y1`: Coordenadas que definen los límites del monitor.
  - `Canvas *canvas`: Puntero al lienzo sobre el cual opera.
  - `my_mutex_t mutex`: Mutex para sincronizar el acceso concurrente.

Funciones principales

- `void monitor_init(Monitor *mon, int x0, int y0, int x1, int y1, Canvas *canvas);`
  Inicializa un monitor con las coordenadas especificadas y lo asocia con un lienzo determinado. También inicializa su mutex.
- `void monitor_draw(Monitor *mon, int current_time);`
  Dibuja en consola todas las figuras dentro de los límites del monitor que estén activas en el momento especificado por `current_time`. El acceso al lienzo se realiza de forma segura mediante el uso de mutex.

implementación

1. `monitor_init`:
   - Asigna los límites `(x0, y0)` y `(x1, y1)` al monitor.
   - Asocia el puntero al `Canvas`.
   - Inicializa el mutex con `my_mutex_init`.
   - Imprime información del lienzo y del monitor para depuración.
2. `monitor_draw`:
   - Bloquea el mutex del monitor para acceder al canvas de forma segura.
   - Recorre todas las figuras en el canvas.
   - Imprime en consola únicamente aquellas figuras cuyas coordenadas actuales estén dentro de los límites del monitor.
   - Libera el mutex al finalizar la lectura.

3.7. `main.c`

El archivo `main.c` contiene el punto de entrada principal del sistema de animación. Se encarga de:

- Leer y parsear el archivo de animación.
- Inicializar el lienzo (`canvas`) y las figuras.

- Crear e inicializar monitores que observarán diferentes regiones del lienzo.
- Configurar y ejecutar el planificador de hilos (scheduler) para simular concurrencia.

Función `monitor_loop`

Función que ejecuta cada hilo-monitor:

- Cada hilo avanza el tiempo global de la animación desde 0 hasta 10.
- En cada iteración, actualiza el estado del lienzo con `canvas_update` y dibuja su porción con `canvas_draw`.
- Hace un `yield` para ceder el CPU y permitir la rotación entre hilos.

Función `main`

Flujo de ejecución:

1. Se parsea el archivo `animation.ani` usando `parse_animation`, almacenando la información en una estructura `Animation`.
2. Se inicializa el lienzo con las dimensiones definidas en la animación.
3. Se convierte cada objeto animado en una figura gráfica usando `object_to_figure`, y se añade al lienzo.
4. Se inicializan los monitores que observarán distintas secciones del lienzo, dividiéndolo equitativamente.
5. Se utiliza `my_thread_create` para crear un hilo por cada monitor.
6. Se inicializa el planificador con la política `ROUND_ROBIN` y se ejecuta con `scheduler_run`.

4. Instrucciones para ejecutar el programa

Para compilar y ejecutar el programa animador, siga los siguientes pasos desde la terminal:

Compilación

```
make clean && make all
```

Ejecución

Ejecute el programa especificando el scheduler a usar:

- Para utilizar **Round Robin**:

  ```
  ./animador rr
  ```

- Para utilizar **Lottery Scheduling**:

  ```
  ./animador lottery
  ```

- Para utilizar **Real-Time Scheduling**:

  ```
  ./animador realtime
  ```

5. Actividades realizadas por estudiante

Cuadro 1: Bitácora de actividades - Manfred

Día	**Actividad**	**Horas**
30/04/2025	Reunión para ver el kickoff, introducción	2
02/05/2025	Clases del diagrama UML	1
07/05/2025	Reunión para ver la división de trabajo	1
09/05/2025 - 12/05/2025	Se trabajan las implementaciones de los tres tipos de schedulers con pthreads	4
24/05/2025	Se hace la definición del lenguaje, parser y función inicial de animación	3
28/05/2025	Se trabaja la inicialización, movimiento y colisión de figuras y las pruebas respectivas	3.5
30/05/2025	Se trabaja en la lógica de implementación de los 3 schedulers para mypthreads y sus pruebas respectivas	3
31/05/2025	Se inicia la redacción de la documentación	2
01/06/2025	Pruebas finales, se finaliza la documentación	3
	Total:	**22.5**

Cuadro 2: Bitácora de actividades - Miguel

Día	**Actividad**	**Horas**
25/04/2025	Reunión para ver el kickoff, introducción	2
26/04/2025	Clases del diagrama UML	1
28/04/2025	Reunión para ver la división de trabajo	1
18/05/2025	Implementación inicial de hilos con ucontext	3
19/05/2025	Implementación del scheduler Round Robin y pruebas básicas	2
24/05/2025	Mutex cooperativo: implementación y testeo inicial	2
26/05/2025	`my_thread_join`, mejoras a `mypthread`, refactor de código	4
27/05/2025	Organización del proyecto y limpieza de archivos	1.5
29/05/2025	Inicio del módulo de animación y estructuras base	2
30/05/2025	Canvas y monitores con animación; pruebas con figuras	5
31/05/2025	Correcciones en parser, integración con `object_to_figure`, Makefile, pruebas de compilación	6
01/06/2025	Finalización de `main`, animación funcional, bitácora y documentación	3.5
	Total:	**33**

6. Autoevaluación

Se logró la reimplementación de la bibblioteca pthreads en mypthreads, se implementaron los 3 tipos de schedulers: round robin, lottery y realtime. Igualmente se definió un lenguaje de animación y se proporciona un archivo de texto que es leído por el parser, para así iniciar los hilos, el canvas y las figuras. Se intentó implementar la animación en múltiples displays pero no se logró en su completitud, ni así tampoco las figuras complejas y rotaciones.

6.1. Evaluación

- **Scheduler RoundRobin: 10**
- **Scheduler Sorteo: 10**
- **Scheduler en Tiempo Real: 10**
- **Cambio de Scheduler: 8**
- **Funciones de la biblioteca pthreads: 9**
- **Documentación en LaTex: 10**
- **Diseño del lenguaje: 8**
- **Implementación de la animación: 7**
- **Funcionamiento en multiples displays: 5**
- **Kick-off: 10**

6.2. Auto-evaluación

- **Aprendizaje de RoundRobin: 5/5**

- **Aprendizaje de Tiempo Real: 5/5**
- **Aprendizaje de cambio de contexto: 4/5**
- **Aprendizaje de sorteo: 5/5**

6.3. Reporte de commits

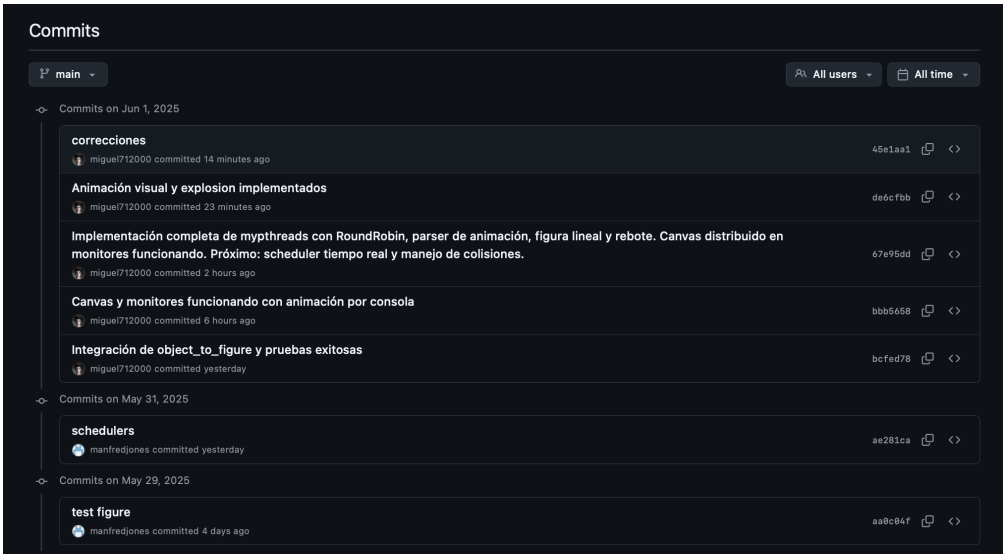

Figura 1: Reporte de commits

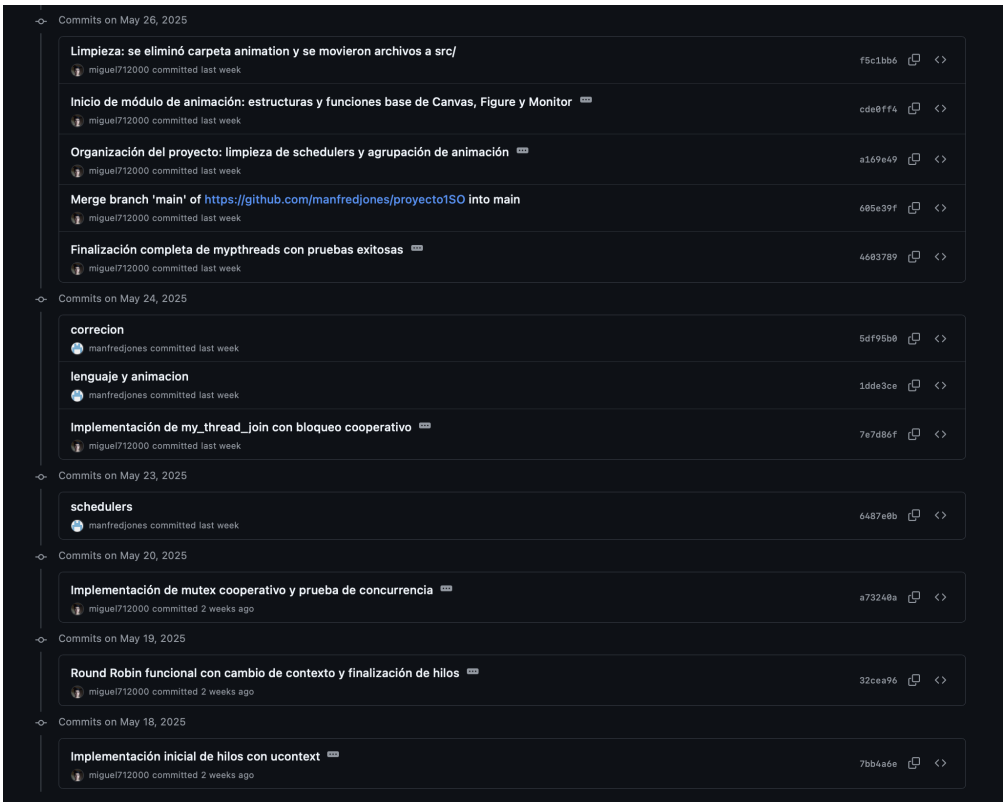

Figura 2: Cont. reporte de commits

7. Lecciones aprendidas

Llevar a cabo este proyecto brindó un aprendizaje importante sobre el manejo de hilos, al reimplementar la biblioteca pthreads desde cero. Además de tener un contacto de primera mano con algoritmos de planificación y cómo estos afectan el desempeño de nuestro programa. Para el desarrollo del lenguaje y parser fue necesario emplear conocimientos de cursos anteriores como lenguajes de programación y estructuras de datos. Se tuvo que tomar decisiones de diseño importantes a la hora de integrar el canvas, figuras e hilos. A pesar de los desafíos y el no poder implementar ciertas funcionalidades clave como los múltiples monitores y las figuras compuestas, este proyecto nos sirvió

Referencias

[1] Andrew S. Tanenbaum, *Modern Operating Systems*, 3ª edición, Pearson, 2009.

[2] IEEE, *POSIX Threads Programming*. Disponible en: https://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_create.html. Accedido el 10 de mayo de 2025.