PROYECTO FINAL.

SISTEMAS OPERATIVOS .

INTEGRANTES:

MAURICIO BELTRÁN HUERTAS

ALEJANDRO BELTRAN HUERTAS

ANDRÉS DÍAZ

2025

1.Objetivos del proyecto

- Resolver un problema real mediante el uso de procesos e hilos de la biblioteca POSIX.
- Emplear mecanismos de sincronización y comunicación entre procesos usando pipes.
- Utilizar correctamente las llamadas al sistema relacionadas con la creación y manejo de hilos y procesos

2.Descripción general del proyecto

En este proyecto se implementa un sistema que permite hacer reservas por horas para entrar a un parque privado. El Parque Berlín es pequeño y durante la época de vacaciones llega mucha gente, lo que hace que el lugar se llene rápidamente y sea difícil organizar el ingreso. Por eso se creó un sistema de control de entrada basado en reservas previas.

La aplicación simula un día completo de funcionamiento del parque, donde diferentes familias intentan reservar una hora para poder entrar. Para esto se diseñó una arquitectura cliente-servidor compuesta por dos tipos de procesos:

- Servidor que en este caso es el controlador de reserva: es el proceso principal que administra cuántas personas pueden entrar, acepta o rechaza reservas, simula el paso del tiempo y genera reportes.
- Clientes que en este caso son los agentes de reserva: son procesos independientes que leen solicitudes desde archivos CSV y se las envían al servidor para que las procese.

La comunicación entre los procesos se realiza usando *pipes* nominales (FIFO), siguiendo los estándares POSIX.

2.1 Servidor: Controlador de Reserva

El servidor recibe las solicitudes enviadas por los agentes y decide si las acepta o las rechaza según el espacio disponible en el parque. También se encarga de simular el paso del tiempo: cada “hora” simulada corresponde a una cantidad configurable de segundos reales.

Cada vez que avanza una hora, el servidor realiza dos acciones:

- Salen las familias que entraron hace dos horas.
- Entran las familias que tienen reserva para la hora actual.

Además, si una reserva llega en un horario que ya está lleno, el servidor intenta reprogramarla buscando el siguiente bloque de tiempo disponible a partir de la hora actual.

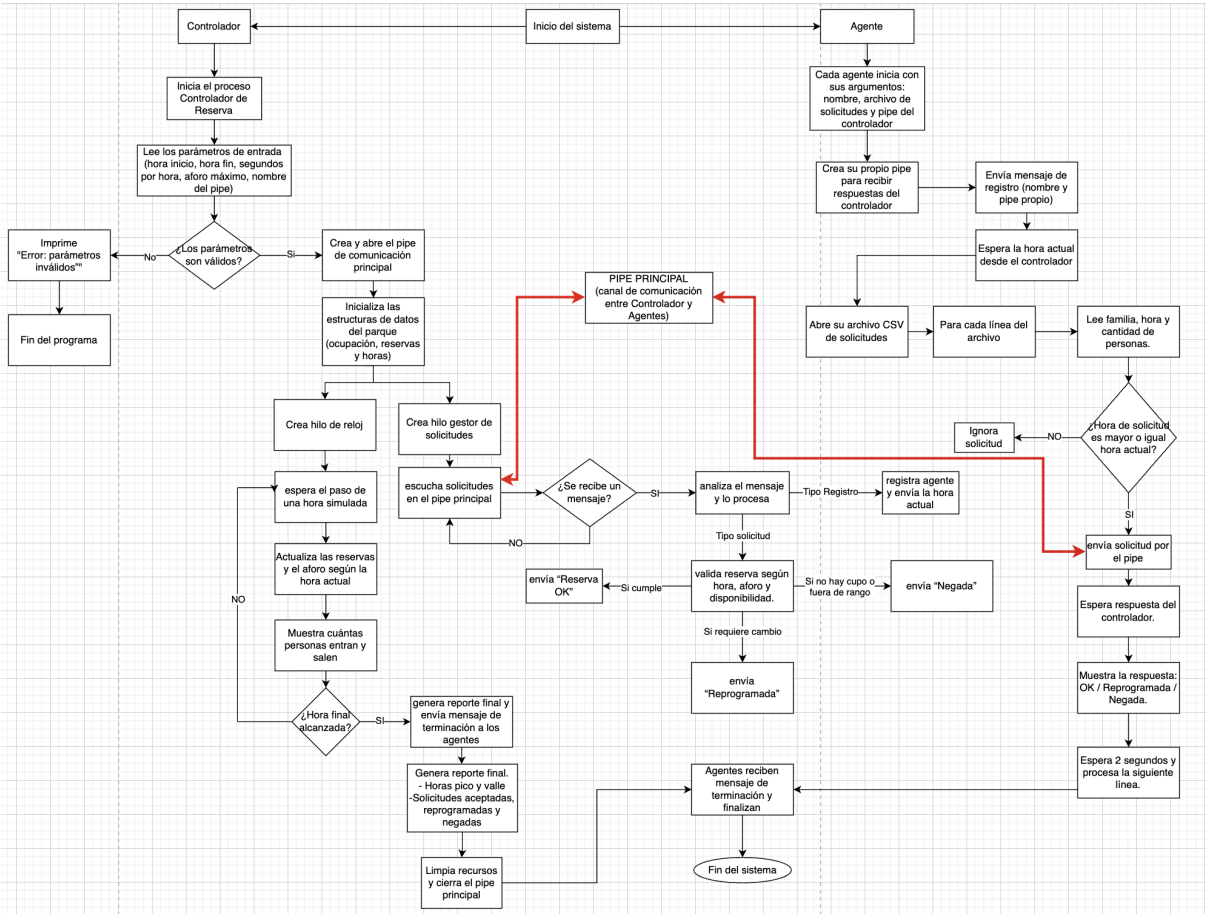
El servidor rechaza una solicitud de inmediato cuando ocurre alguna de estas condiciones:

- El número de personas supera el aforo permitido.
- La hora solicitada está por fuera del horario de funcionamiento.
- No hay espacio disponible, ni siquiera moviendo la reserva a otro horario.

2.2. Cliente: Agente de Reserva

- El cliente lee las solicitudes desde un archivo CSV con el formato: *familia, hora, personas*.
- Ignora las solicitudes cuya hora ya pasó en relación con la hora simulada actual. Luego, envía cada solicitud al servidor y espera la respuesta correspondiente.
- El cliente muestra en pantalla el resultado que reciba del servidor, que puede ser: *OK, REPROGRAMADA, NEGADA o NEGADA_EXT.*
  Cuando termina de leer todas las solicitudes del CSV, cierra su pipe y finaliza su ejecución.

3. Arquitectura del sistema



Como se puede observar no basta con iniciar uno solo de los programas, es necesario iniciar tanto el controlador como el agente. En el primero, recibe varios parámetros de entrada como la hora de inicio y fin de la simulación, la cantidad de segundos en una hora simulada, el aforo máximo del parque y el nombre del pipe que va a usar para comunicarse. Una vez ingresados ha de revisar si coinciden con lo esperado en cuanto a formato y cantidad, en caso de ser válidos abre el pipe principal e inicializa el parque, las reservas y las horas, estas últimas tienen que ir avanzando como un reloj, este proceso tanto por organización como funcionalidad está a cargo de un hilo así que se crea. Por otra parte, las solicitudes también son controladas desde un hilo encargado del pipe del sistema por donde pasan los mensajes de los agentes[1], sin bloquear o afectar el funcionamiento del controlador, debe recibir los mensajes, registrar los agentes y sus solicitudes junto a procesar las mismas.

De igual manera, el lado del agente debe ir avanzando primero desde el comando de ejecución recibe los parámetros desde su nombre, nombre del archivo con los datos de la solicitud hasta el pipe del controlador que va a utilizar. Posteriormente, crea un pipe propio para recibir respuestas y envía un primer mensaje en el que indica su nombre y el pipe en el que espera las respuestas. Adicionalmente, recibe la hora actual en el parque de simulación y abre el archivo de datos lee el nombre de la familia, la hora de la reserva y la cantidad de personas en la familia, si la hora ya pasó ignora esa reserva. De lo contrario, la envía través

del pipe del controlador donde es recibida por el hilo de solicitudes, inicialmente recibe el mensaje de registro del agente en donde lo registra y le envía la hora actual, de esta manera el agente revisa lo previamente mencionado [2].

La siguiente parte, consiste en revisar la solicitud, como primer filtro mira la cantidad de integrantes en la familia para mirar si al sumarlos no se exceda el aforo del parque de excederse esta solicitud es negada. Asimismo, mira la hora de la reserva y considera su duración (2 horas) si no hay tiempo para ésta la niega, o en que caso de que ya pasará pero sigue habiendo disponibilidad la reprograma, la otra alternativa es que el espacio está en la hora solicitada por lo que la acepta. Posteriormente, el agente recibe la respuesta a la solicitud, espera segundos en caso de que haya enviado más de una solicitud espera por más respuestas y una vez terminada su función cierra los puertos para leer los archivos, el pipe propio para respuestas y se finaliza.

Mientras tanto, el proceso continúa del lado del controlador en donde empieza a pasar el tiempo (avanzando las horas), actualizando las reservas, las personas que entran y salen del parque, el aforo total hasta llegar a la hora de cierre. Finalmente, envía un mensaje de terminación para los agentes, genera un reporte de la simulación, con hora pico, hora valle, cuantas solicitudes acepto, reprogramo y rechazo, también cierra los recursos que ya no van a ser utilizados como su pipe y se termina.

4. Descripciones

4.1 controlador.c

Este archivo cumple el rol fundamental de manejar la simulación completa del sistema de reservas. Desde la lectura y validación de parámetros hasta la inicialización del parque y la creación de los hilos, su función es garantizar que todos los elementos del proyecto operen en coherencia. La correcta configuración del entorno, incluyendo pipes, estructuras de horas, límites de aforo y tiempos de simulación, permite que los módulos de gestión y reloj trabajen de manera estable y sincronizada. Además, la creación de hilos independientes asegura concurrencia real, permitiendo que el controlador responda dinámicamente a las solicitudes de los agentes. Finalmente, este archivo establece la base operativa necesaria para que el sistema funcione de manera ordenada, modular y eficiente, logrando así los objetivos del proyecto en un entorno concurrente.

4.2 controlador_funciones.h

Este archivo centraliza todas las estructuras de datos y prototipos necesarios para el funcionamiento del controlador. Su organización modular permite separar claramente la gestión del reloj, las reservas, la comunicación por pipes y la sincronización entre hilos. La definición explícita de constantes, tipos y variables globales facilita la integración entre los diferentes componentes del proyecto y asegura un manejo ordenado de la simulación. Gracias a esta arquitectura, el controlador puede coordinar múltiples agentes, validar aforos, reprogramar reservas y mantener la consistencia del sistema en tiempo real.

4.3 controlador_funciones.c

Este archivo genera el motor del controlador, ejecutando las funciones críticas que permiten la simulación completa del parque. El uso de hilos independientes para el reloj y la gestión de solicitudes garantiza concurrencia realista y evita bloqueos entre las distintas entidades del sistema. La implementación del algoritmo de reservas, con validación de aforo, detección de conflictos y búsqueda de bloques disponibles, asegura un manejo eficiente y coherente de las solicitudes. Igualmente, la comunicación mediante pipes permite una interacción clara y ordenada entre agentes y controlador. Finalmente, el reporte generado al concluir la simulación ofrece una visión global del comportamiento del sistema, evidenciando horas pico, horas valle y estadísticas clave de aceptación, reprogramación y negación de reservas. Con esto, se cumple exitosamente el objetivo del proyecto: modelar un entorno concurrente estable, sincronizado y funcional.

4.4agente.c

Este archivo implementa el flujo principal de ejecución del agente, integrando todas las funciones necesarias para su comunicación con el controlador. Desde el procesamiento de argumentos hasta el cierre de recursos, el agente opera de manera estructurada y autónoma, garantizando que cada solicitud sea enviada y procesada correctamente. Al coordinar la creación del pipe propio, el registro, la recepción de la hora inicial y el envío de solicitudes, el agente cumple su rol dentro del sistema concurrente, funcionando como un componente activo y sincronizado con el controlador. En conjunto, este archivo asegura un funcionamiento ordenado, modular y eficiente del agente, permitiendo ejecutar la simulación de manera confiable y conforme a los objetivos del proyecto.

4.5 agente_funciones.h

Este archivo establece la interfaz fundamental del agente, proporcionando los prototipos y parámetros necesarios para su comunicación con el controlador. Al definir de manera clara las funciones de registro, envío de solicitudes, recepción de respuestas y manejo de recursos, se garantiza una interacción ordenada, modular y confiable dentro del sistema concurrente.

Gracias a esta estructura, los agentes pueden operar de forma independiente mientras mantienen sincronización con el controlador, permitiendo así el funcionamiento distribuido y estable de la simulación.

4.6 agente_funciones.c

Este archivo implementa la lógica completa del agente, permitiendo su participación activa en el sistema de reservas. A través del manejo correcto de pipes, lectura de archivos CSV, validación de solicitudes y recepción de respuestas, el agente logra comunicarse de forma ordenada y confiable con el controlador. La modularidad presentada aquí permite que cada agente funcione de manera independiente, garantizando la concurrencia y evitando interferencias entre ellos. Asimismo, los procesos de registro, envío de mensajes, espera de respuestas y limpieza final aseguran un flujo de ejecución claro y estable durante toda la simulación. Con este diseño, el agente cumple adecuadamente su rol de intermediario entre los usuarios y el controlador, contribuyendo al funcionamiento armónico del sistema.

4.7 makefile

Este Makefile facilita la gestión del proyecto al automatizar las tareas de compilación y limpieza de archivos auxiliares. Gracias a su estructura clara, permite compilar rápidamente tanto el controlador como los agentes, asegurando que todos los módulos se construyen con los flags y dependencias adecuadas. Además, la inclusión de la regla clean ayuda a mantener el entorno de trabajo libre de ejecutables y pipes residuales, garantizando que la simulación pueda reiniciarse sin conflictos. De esta manera, el Makefile contribuye de forma esencial a la organización y correcta ejecución del sistema desarrollado.

5. Plan de pruebas

Caso de prueba	Descripción	Comando(s) ejecutados	Resultado esperado	Resultado obtenido
Prueba-01	Ejecución solo del controlador	./controlador -i 7 -f 12 -s 2 -t 10 -p pipeControlador	El controlador arranca, crea el pipe y muestra avance de horas, pero no procesa solicitudes.	El controlador arranca, crea el pipe y muestra avance de horas, pero no procesa solicitudes.
Prueba-02	Ejecución del controlador + un agente con archivo solicitudes1.csv	1) ./controlador -i 7 -f 12 -s 2 -t 10 -p pipeControlador2) ./agente -s Agente1 -a solicitudes1.csv -p pipeControlador	El agente se registra, se envían solicitudes, el controlador acepta, reprograma o rechaza según aforo.	El agente se registra, se envían solicitudes, el controlador acepta, reprograma o rechaza según aforo.
Prueba-03	Ejecución del controlador + mismo agente pero con archivo solicitudes2.csv	1) ./controlador -i 7 -f 12 -s 2 -t 10 -p pipeControlador2) ./agente -s Agente1 -a solicitudes2.csv -p pipeControlador	El sistema debe procesar correctamente un archivo distinto, manteniendo consistencia en respuestas.	El sistema debe procesar correctamente un archivo distinto, manteniendo consistencia en respuestas.
Prueba-04	Ejecución del controlador + dos agentes simultáneos	1) ./controlador -i 7 -f 12 -s 2 -t 10 -p pipeControlador2) ./agente -s Agente1 -a solicitudes1.csv -p pipeControlador3) ./agente -s Agente2 -a solicitudes2.csv -p pipeControlador	Ambos agentes deben registrarse, enviar solicitudes y recibir respuestas sin interferencia. El controlador debe manejar la concurrencia correctamente.	Ambos agentes deben registrarse, enviar solicitudes y recibir respuestas sin interferencia. El controlador debe manejar la concurrencia correctamente.
Prueba-05	Ejecución del controlador + tres agentes simultáneos	1) ./controlador -i 7 -f 12 -s 2 -t 10 -p pipeControlador2) ./agente -s Agente1 -a solicitudes1.csv -p pipeControlador3) ./agente -s Agente2 -a solicitudes2.csv -p pipeControlador4) ./agente -s Agente3 -a solicitudes3.csv -p pipeControlador	El controlador debe procesar múltiples agentes concurrentes, sin bloqueos ni pérdida de mensajes.	El controlador debe procesar múltiples agentes concurrentes, sin bloqueos ni pérdida de mensajes.

Con el fin de validar el correcto funcionamiento del programa se plantearon 5 pruebas cada una con un fin específico. Asimismo, se documentaron la situación que se estaba ejecutando los comandos utilizados para ejecutar, el resultado obtenido y el resultado esperado. Cabe mencionar, que para cada una de estas pruebas se añade pantallazos de ejecución como evidencia y el objetivo de su realización.

Prueba-01:

Con el fin de verificar que el controlador inicializa correctamente (incluyendo el pipe principal, variables y simulación de tiempo), incluso cuando no existen agentes conectados.

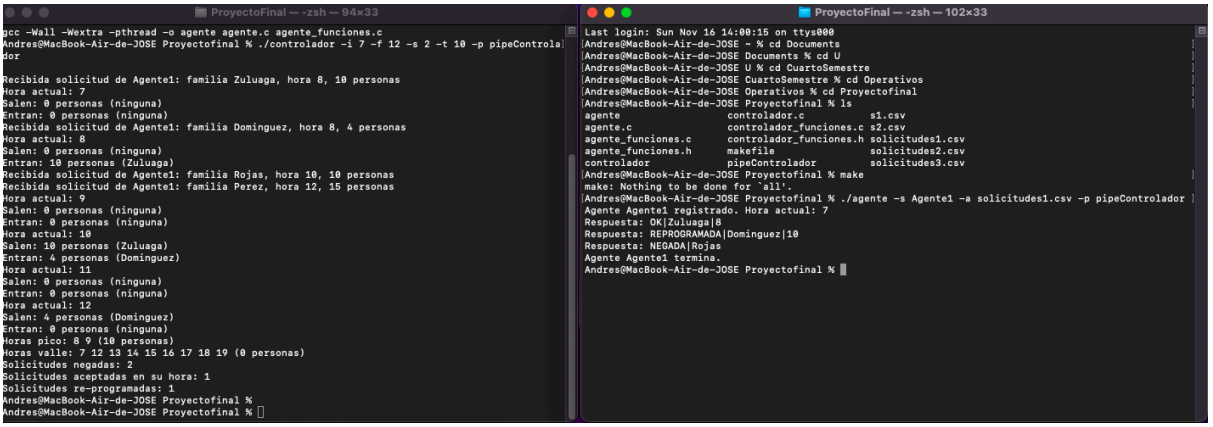
Evidencia:

```
[Andres@MacBook-Air-de-JOSE Proyectofinal % make
gcc -Wall -Wextra -pthread -o controlador controlador.c controlador_funciones.c
gcc -Wall -Wextra -pthread -o agente agente.c agente_funciones.c
[Andres@MacBook-Air-de-JOSE Proyectofinal % ./controlador -i 7 -f 12 -s 2 -t 10 -p pipeControlador
```

Prueba-02:

Verificar la comunicación básica entre un agente y el controlador, confirmando que el envío de solicitudes, recepción de respuestas y la gestión del aforo se realizan correctamente.

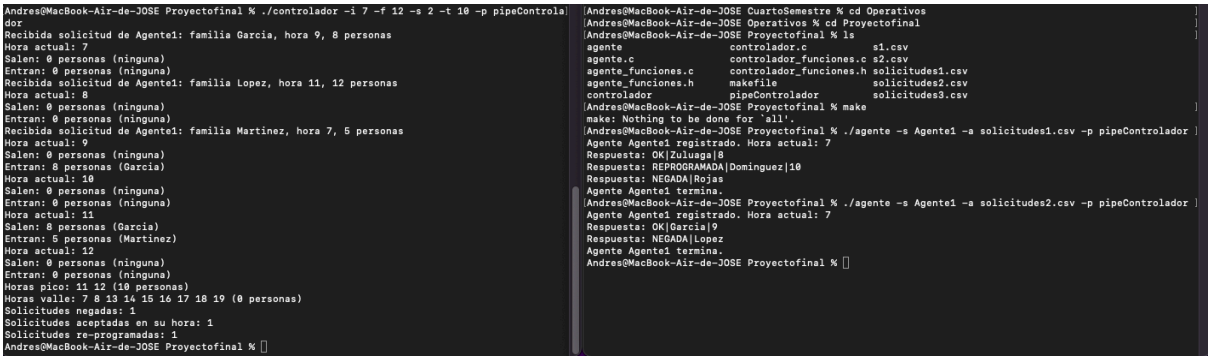
Evidencia:



Prueba-03:

Evaluar la flexibilidad y consistencia del sistema frente a diferentes fuentes de entrada, comprobando que el controlador procesa correctamente un archivo distinto.

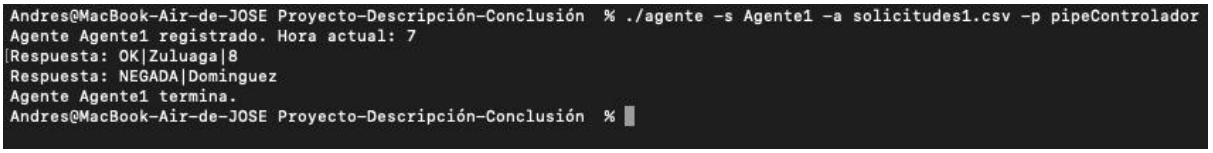
Evidencia:

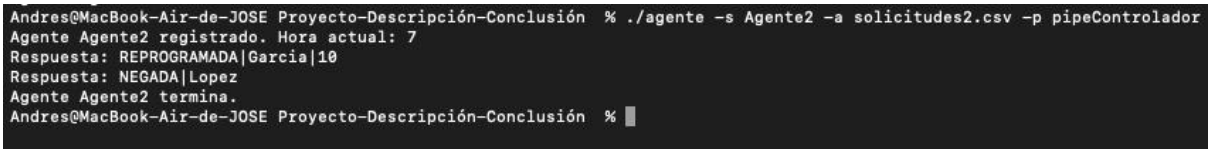


Prueba-04:

Validar la correcta sincronización y concurrencia entre múltiples agentes, asegurando que los mensajes no se pierdan, no existan colisiones en el pipe y que el controlador mantenga control del aforo global con precisión.

Evidencia:



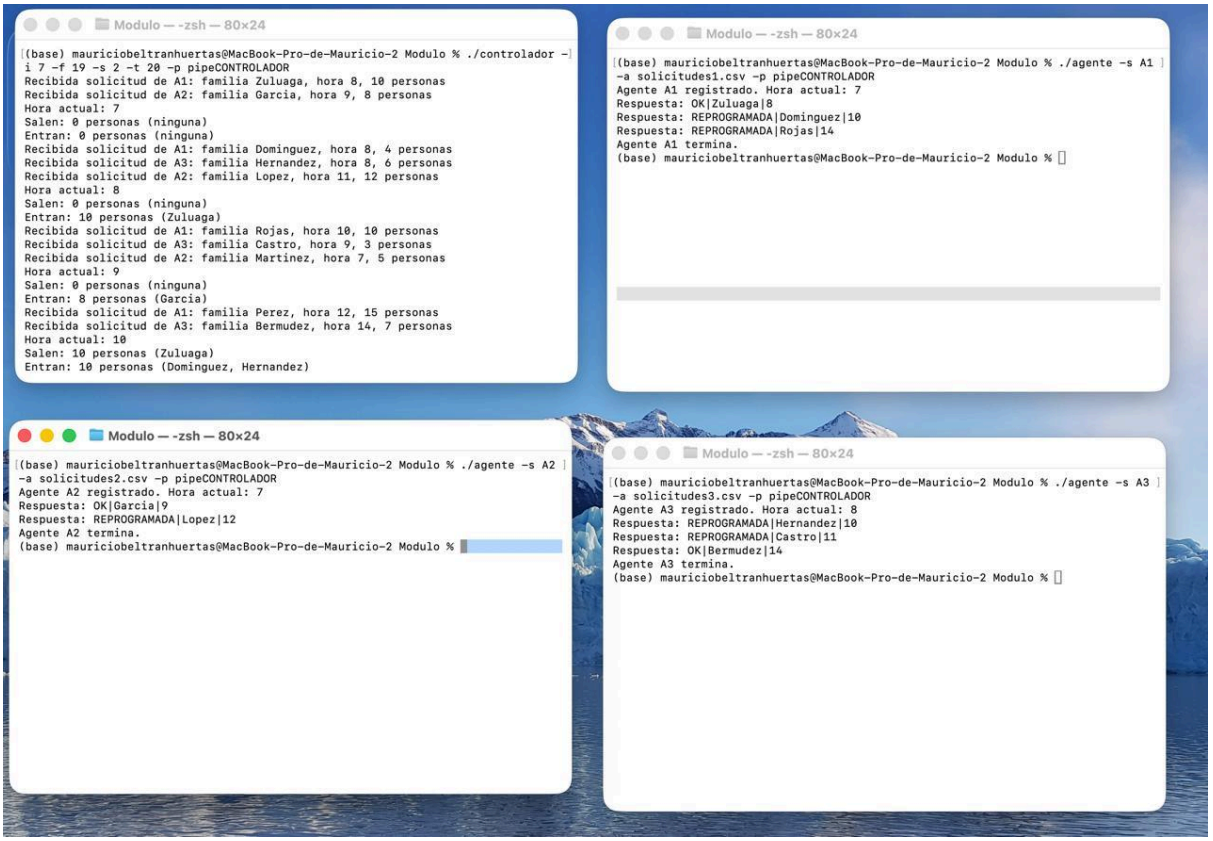


```
Andres@MacBook-Air-de-JOSE Proyecto-Descripción-Conclusión  % ./controlador -i 7 -f 12 -s 2 -t 10 -p pipeControlador
[Recibida solicitud de Agente1: familia Zuluaga, hora 8, 10 personas
Recibida solicitud de Agente2: familia Garcia, hora 9, 8 personas
Hora actual: 7
Salen: 0 personas (ninguna)
Entran: 0 personas (ninguna)
Recibida solicitud de Agente1: familia Dominguez, hora 8, 4 personas
Recibida solicitud de Agente2: familia Lopez, hora 11, 12 personas
Hora actual: 8
Salen: 0 personas (ninguna)
Entran: 10 personas (Zuluaga)
Recibida solicitud de Agente1: familia Rojas, hora 10, 10 personas
Recibida solicitud de Agente2: familia Martinez, hora 7, 5 personas
Hora actual: 9
Salen: 0 personas (ninguna)
Entran: 0 personas (ninguna)
Hora actual: 10
Salen: 10 personas (Zuluaga)
Entran: 8 personas (Garcia)
Hora actual: 11
Salen: 0 personas (ninguna)
Entran: 0 personas (ninguna)
Hora actual: 12
Salen: 8 personas (Garcia)
Entran: 0 personas (ninguna)
Horas pico: 8 9 (10 personas)
Horas valle: 7 12 13 14 15 16 17 18 19 (0 personas)
Solicitudes negadas: 4
Solicitudes aceptadas en su hora: 1
Solicitudes re-programadas: 1
Andres@MacBook-Air-de-JOSE Proyecto-Descripción-Conclusión  %
```

Prueba-05:

Validar la correcta sincronización y concurrencia entre múltiples agentes, asegurando que los mensajes no se pierdan, no existan colisiones en el pipe y que el controlador mantenga control del aforo global con precisión. En una situación de mayor carga al utilizar 3 agentes.

Evidencia:



Datos de los archivos de entrada utilizados:

solicitudes1.csv

solicitudes1.csv

Zuluaga,8,10

Dominguez,8,4

Rojas,10,10

Perez,12,15

solicitudes2.csv

solicitudes2.csv

Garcia,9,8

Lopez,11,12

Martinez,7,5

solicitudes3.csv

solicitudes3.csv

Hernandez 8 , 6

Castro 9 , 3

Bermudez 14 , 7

Rincon 15 , 4

6. Conclusiones

El programa logra simular un sistema de reservas para un parque, considerando reservas aceptadas, negadas y re programas, considerando el horario de este, la duración de cada una de las reservas y el aforo del parque.

De igual manera, se evitaron condiciones de carrera mediante el bloqueo de pipes y una gran separación de responsabilidades para el buen funcionamiento del programa, factor fundamental ya que en este simulador existe un controlador y uno o más agentes cada uno con sus propias reservas y enviando sus datos. Es decir, todas las entidades funcionan correctamente sin bloquearse las unas a las otras o producir un overhead, distribuyendo apropiadamente los recursos y permitiendo la simulación.

Adicionalmente, se implementaron distintos hilos con funciones como revisar el contenido de los pipes o avanzar el reloj esta distribución de funciones permite tener un programa más ordenado y sincronizado. Finalmente, el código tiene una estructura modular que permite ubicar las funciones, variables que este maneja y su funcionamiento, sumado a los comentarios presentes en cada uno de los archivos se vuelve aún más claro y fácil de replicar.

LINK REPOSITORIO PERSONAL:

https://github.com/Mauriciobh05/ProyectoOperativoFinal

7. Referencias

[1] *Handling multiple clients on server with multithreading using Socket Programming in C/C++*. (2021, agosto 10). GeeksforGeeks. https://www.geeksforgeeks.org/cpp/handling-multiple-clients-on-server-with-multithreading-using-socket-programming-in-c-cpp/

[2] Kalin, M. (n.d.). *Inter-process communication in Linux: Sockets and signals*. Opensource.com. Retrieved November 18, 2025, from

https://opensource.com/article/19/4/interprocess-communication-linux-networking