Simulación de un Sistema de Reservas

David Tobar Artunduaga, Daniel Ramírez & Guillermo Aponte

Pontificia Universidad Javeriana

Facultad de Ingeniería

Sistemas Operativos

Bogotá, D.C

.

18/11/2025

Tabla de Contenidos

Objetivos

Objetivo General

Diseñar e implementar un sistema de reservas concurrente para un parque, empleando procesos, hilos y mecanismos de comunicación entre procesos en Linux (named pipes), que permita simular y gestionar de forma correcta el aforo, las solicitudes de reserva y el flujo de usuarios en un rango de horas determinado.

Objetivos Específicos

- Crear el proceso servidor y los procesos agentes usando POSIX.
- Implementar la comunicación entre procesos mediante pipes con nombre (mkfifo, open, read, write).
- Simular el paso del tiempo usando un hilo que avance la hora según el parámetro segHora
- Procesar y responder solicitudes de reserva validando aforo, hora solicitada y disponibilidad.
- Registrar en consola las solicitudes recibidas y las respuestas enviadas a cada agente
- Generar un reporte final con horas pico, horas de menor ocupación y estadísticas de reservas.
- Validar errores en todas las llamadas al sistema y cerrar correctamente todos los archivos usados para el desarrollo del proyecto y liberar la memoria reservada.

Resumen

El proyecto implementa un sistema de reservas concurrente para un parque, utilizando procesos POSIX, hilos y comunicación mediante named pipes en Linux. El sistema sigue una arquitectura cliente-servidor: el Controlador gestiona las solicitudes de reserva, simula el paso del tiempo y administra el aforo por hora, mientras que los Agentes leen solicitudes desde un archivo y las envían al Controlador, esperando su respuesta.
El documento explica los conceptos teóricos necesarios (procesos, hilos, mutex, semáforos, memoria dinámica y llamadas al sistema) y describe la implementación modular del Controlador y los Agentes, incluyendo validación de parámetros, manejo del reloj simulado, recepción de mensajes y generación de reportes. Asimismo, presenta un plan de pruebas que verifica el comportamiento del sistema bajo diferentes condiciones, como errores de entrada, solicitudes inválidas y concurrencia de múltiples agentes.

Marco Teórico

Intercomunicación de Procesos

(Silberschatz et al., 2020) “Los procesos que se ejecutan simultáneamente en el sistema operativo pueden ser independientes o cooperantes. Un proceso es independiente si no comparte datos con otros procesos que se ejecutan en el sistema. Un proceso coopera si puede afectar o ser afectado por los demás procesos que se ejecutan en el sistema. Claramente, cualquier proceso que comparte datos con otros procesos es un proceso cooperante.”

Proceso

(Silberschatz et al., 2020) “Un proceso es un programa en ejecución. Un proceso necesita ciertos recursos, como tiempo de CPU, memoria, archivos y dispositivos de E/S, para realizar su tarea. Estos recursos suelen asignarse al proceso mientras se ejecuta. Un proceso es la unidad de trabajo en la mayoría de los sistemas. Los sistemas constan de un conjunto de procesos: los procesos del sistema operativo ejecutan el código del sistema y los procesos de usuario ejecutan el código del usuario. Todos estos procesos pueden ejecutarse simultáneamente. Los sistemas operativos modernos admiten procesos con múltiples hilos de control. En sistemas con múltiples núcleos de procesamiento de hardware, estos hilos pueden ejecutarse en paralelo. 3 Uno de los aspectos más importantes de un sistema operativo es cómo programa los hilos en los núcleos de procesamiento disponibles. Los programadores disponen de varias opciones para diseñar programadores de CPU.”

Semáforos

(*sem_overview(7) - Linux manual page*, s. f.) “Los semáforos POSIX permiten que los procesos e hilos sincronicen sus acciones. Un semáforo es un entero cuyo valor nunca puede ser inferior a cero. Se pueden realizar dos operaciones con los semáforos: incrementar su valor en uno (sem_post(3)) y decrementarlo en uno (sem_wait(3)). Si el valor de un semáforo es cero, la operación sem_wait(3) se bloqueará hasta que el valor sea mayor que cero. Los semáforos POSIX se presentan en dos formas: semáforos con nombre y semáforos sin nombre.”

Semáforos con Nombre

(*sem_overview(7) - Linux manual page*, s. f.) “Un semáforo con nombre se identifica mediante un nombre con el formato /nombre; es decir, una cadena terminada en nulo de hasta NAME_MAX-4 (es decir, 251) caracteres que consta de una barra inicial seguida de uno o más caracteres, ninguno de los cuales es una barra. Dos procesos pueden operar sobre el mismo semáforo con nombre pasando el mismo nombre a sem_open()”.

Memoria Compartida POSIX

(*shm_overview(7) - Linux manual page*, s. f.) La API de memoria compartida POSIX permite que los procesos comuniquen información compartiendo una región de memoria"

Función sem_open()

(*sem_open(3) - Linux manual page*, s. f.) "sem_open() crea un nuevo semáforo POSIX o abre uno existente."

Función sem_post()

(*sem_post(3) - Linux manual page*, s. f.) "`sem_post()` incrementa (desbloquea) el semáforo al que apunta `sem`. Si el valor del semáforo pasa a ser mayor que cero, entonces otro proceso o hilo bloqueado en una llamada a `sem_wait(3)` se reactivará y procederá a bloquear el semáforo."

Función sem_wait()

(*sem_wait(3) - Linux manual page*, s. f.) "`sem_wait()` decrementa (bloquea) el semáforo al que apunta `sem`. Si el valor del semáforo es mayor que cero, entonces el decremento se realiza y la función retorna inmediatamente. Si el semáforo actualmente tiene el valor cero, la llamada se bloquea hasta que sea posible realizar el decremento (es decir, el valor del semáforo supere cero) o un manejador de señales interrumpa la llamada."

Función sem_close()

(*sem_close(3) - Linux manual page*, s. f.) "sem_close() cierra el semáforo especificado por sem, permitiendo que se liberen todos los recursos que el sistema haya asignado al proceso que realiza la llamada para este semáforo."

Función sem_unlink()

(*sem_unlink(3) - Linux manual page*, s. f.) "sem_unlink() elimina el semáforo especificado por su nombre. El nombre del semáforo se elimina inmediatamente. El semáforo se destruye una vez que todos los demás procesos que lo tienen abierto lo cierran."

Función unlink()

(*unlink(2) - Linux manual page*, s. f.) "La función `unlink()` elimina un nombre del sistema de archivos. Si ese nombre era el último enlace a un archivo y ningún proceso lo tiene abierto, el archivo se elimina y el espacio que utilizaba queda disponible para su reutilización. Si el nombre

era el último enlace a un archivo, pero algún proceso aún lo tiene abierto, el archivo permanecerá en el sistema hasta que se cierre el último descriptor de archivo que lo referencia."

Mutex

(AI-FutureSchool, 2025) "Un mutex, o mutua esclusión, es un mecanismo de sincronización utilizado en programación concurrente para gestionar el acceso a recursos compartidos entre múltiples hilos de ejecución. Su principal objetivo es prevenir condiciones de carrera, donde dos o más hilos intentan acceder y modificar un recurso al mismo tiempo, lo que puede resultar en comportamientos inesperados y errores en el programa."

Cond

(*z/OS*, 2021) "Se bloquea en una variable de condición. Debe llamarse con el mutex bloqueado por el hilo que realiza la llamada; de lo contrario, se producirá un comportamiento indefinido. Un mutex se bloquea mediante pthread_mutex_lock(). cond es una variable de condición que comparten los hilos. Para cambiarla, un hilo debe mantener el mutex asociado a la variable de condición. La función pthread_cond_wait() libera este mutex antes de suspender el hilo y lo obtiene de nuevo antes de regresar."

Función pthread_create()

(*pthread_create(3) - Linux manual page*, s. f.) "La función pthread_create() inicia un nuevo hilo en el proceso que la llama."

Función pthread_join()

(*pthread_join(3) - Linux manual page*, s. f.) "La función pthread_join() espera a que finalice el hilo especificado por thread. Si dicho hilo ya ha finalizado, entonces pthread_join() devuelve el control inmediatamente. El hilo especificado por thread debe ser unible para la unión de hilos."

Función pthread_mutex_lock()

(*AIX*, 2025) "El objeto de mútex al que hace referencia el parámetro mutex se bloquea llamando a pthread_mutex_lock. Si el mutex ya está bloqueado, la hebra de llamada se bloquea hasta que el mutex esté disponible. Esta operación se devuelve con el objeto mutex al que hace referencia el parámetro mutex en el estado bloqueado con la hebra de llamada como su propietario."

Función pthread_mutex_unlock()

(*Z/OS*, 2023) "Libera un objeto mutex. Si uno o más hilos están esperando para bloquear el mutex, pthread_mutex_unlock() hace que uno de esos hilos regrese de pthread_mutex_lock() con el objeto mutex adquirido. Si ningún hilo está esperando el mutex, el mutex se desbloquea sin propietario actual"

Función pthread_cond_wait()

(*pthread_cond_wait*, s. f.) “Las funciones pthread_cond_wait() y pthread_cond_timedwait() se utilizan para bloquearse en una variable de condición. Se llaman con el mutex bloqueado por el hilo que realiza la llamada; de lo contrario, se producirá un comportamiento indefinido. Estas funciones liberan el mutex de forma atómica y hacen que el hilo que realiza la llamada se bloquee en la variable de condición cond ; atómicamente aquí significa "atómicamente con respecto al acceso de otro hilo al mutex y luego a la variable de condición". Es decir, si otro hilo puede adquirir el mutex después de que el hilo que está a punto de bloquearse lo haya liberado, entonces una llamada posterior 10 a pthread_cond_signal() o pthread_cond_broadcast() en ese hilo se comporta como si se hubiera emitido después de que el hilo que está a punto de bloquearse se haya bloqueado”.

Función pthread_cond_signal()

(*pthread_cond_signal*, s. f.) “Se utiliza para desbloquear hilos bloqueados en una variable de condición. La llamada a pthread_cond_signal() desbloquea al menos uno de los hilos que están bloqueados en la variable de condición especificada cond (si algún hilo está bloqueado en cond). Si más de un hilo está bloqueado en una variable de condición, la política de planificación determina el orden en que se desbloquean los hilos. Cuando cada hilo desbloqueado como resultado de una llamada a ` pthread_cond_signal() ` o `pthread_cond_broadcast()` regresa de su llamada a `pthread_cond_wait()` o `pthread_cond_timedwait()`”

Memoría Dinámica

(*Manejo de Punteros y Memoria Dinámica | Sistemas Operativos - UTN FRBA*, s. f.) “La memoria dinámica es memoria que se reserva en tiempo de ejecución y se aloja en el heap del proceso (los datos apuntados por los punteros). En C, el programador deberá reservar dicha memoria para su uso y también tendrá la responsabilidad de liberarla cuando no la utilice más. Una diferencia importante es que el tamaño de la memoria dinámica se puede ir modificando durante la ejecución del programa. ¿Qué quiere decir ésto? Que, por ejemplo, podrías ir agrandando/achicando una determinada estructura (por ejemplo, un array) a medida que lo necesitas. Algunas ventajas que ofrece la memoria dinámica frente a la memoria estática es que podemos reservar espacio para variables de tamaño no conocido hasta el momento de la ejecución (por ejemplo, para listas o arrays de tamaños variables), o bloques de memoria que, mientras mantengamos alguna referencia a él, pueden sobrevivir al bloque de código que lo creó.”

Función malloc()

(*malloc(3) - Linux manual page*, s. f.) “La función malloc() asigna size bytes y devuelve un puntero a la memoria asignada. La memoria no se inicializa. Si size es 0, entonces malloc() devuelve un valor de puntero único que posteriormente se puede pasar correctamente a free().

(Consulte "Comportamiento no portátil" para obtener más información sobre problemas de portabilidad).”

Función free()

(*malloc(3) - Linux manual page*, s. f.) “La función `free()` libera el espacio de memoria al que apunta `p`, que debe haber sido devuelto por una llamada previa a `malloc()` o funciones relacionadas. De lo contrario, o si `p` ya se ha liberado, se produce un comportamiento indefinido. Si `p` es `NULL`, no se realiza ninguna operación.”

Función sleep()

(*sleep(3) - Linux manual page*, s. f.) “hace que el hilo que realiza la llamada se suspenda hasta que el número de segundos en tiempo real especificado haya transcurrido o hasta que llegue una señal que no se ignore”

Función open()

(*open(2) - Linux manual page*, s. f.) “La llamada al sistema `open()` abre el archivo especificado por `path`. Si el archivo especificado no existe, `open()` puede crearlo opcionalmente (si se especifica `O_CREAT` en `flags`).

El valor devuelto por `open()` es un descriptor de archivo, un entero pequeño no negativo que es un índice a una entrada en la tabla de descriptores de archivo abiertos del proceso. El descriptor de archivo se utiliza en llamadas al sistema posteriores (`read(2)`, `write(2)`, `lseek(2)`, `fcntl(2)`, etc.) para referirse al archivo abierto. El descriptor de archivo devuelto por una llamada exitosa será el descriptor de archivo con el número más bajo que no esté abierto actualmente para el proceso.”

Función read()

(*read(2) - Linux manual page*, s. f.) “read() intenta leer hasta count bytes desde el descriptor de archivo fd en el búfer que comienza en buf.”

Función write()

(*write(2) - Linux manual page*, s. f.) “write() escribe hasta count bytes desde el búfer que comienza en buf en el archivo al que hace referencia el descriptor de archivo fd.”

Función close()

(*close(2) - Linux manual page*, s. f.) “close() cierra un descriptor de archivo, de modo que ya no hace referencia a ningún archivo y puede reutilizarse.”

Función mkfifo()

(*mkfifo(3) - Linux manual page*, s. f.) “mkfifo() crea un archivo especial FIFO con el nombre path. El modo especifica los permisos del FIFO. Estos se modifican mediante la máscara de permisos (umask) del proceso de la forma habitual.”

Función fopen()

(*fopen(3) - Linux manual page*, s. f.) “La función fopen() abre el archivo cuyo nombre es la cadena a la que apunta la ruta y le asocia un flujo de datos.”

Función fgets()

(*fgets(3p) - Linux manual page*, s. f.) “La función fgets() leerá bytes del flujo en la matriz a la que apunta s hasta que se lean n-1 bytes, o se lea un <nuevo salto de línea> y se transfiera a s, o se encuentre una condición de fin de archivo.”

Descripción de la implementación

Agente

El Agente de Reservas está compuesto por tres funciones principales, cada una encargada de una etapa específica dentro del ciclo de vida del proceso. Estas funciones permiten gestionar la toma de argumentos, la comunicación con el Controlador de Reservas y el procesamiento del archivo de solicitudes.

Función tomarArgumentosAgente()

Se encarga de validar los parámetros con los que se invoca el Agente desde la línea de comandos. Esta función analiza las banderas entregadas por el usuario (-s, -a y -p), verifica que todas estén presentes y que cuenten con un valor válido, y posteriormente almacena dicha información en una estructura RetornoAgentes. En caso de inconsistencias o ausencia de parámetros obligatorios, la función informa el error y retorna un estado inválido. Su propósito principal es garantizar que el Agente disponga de toda la información necesaria antes de iniciar.

Función leerArchivo()

La función leerArchivo concentra la mayor parte de la lógica del Agente. Inicialmente, establece la comunicación con el Controlador abriendo el named pipe principal indicado por los argumentos. De igual forma, crea un named pipe propio para recibir las respuestas del Controlador. Una vez establecidas las conexiones, envía un mensaje inicial con su nombre, permitiendo que el Controlador registre al Agente y le devuelva la hora actual de la simulación. Posteriormente, abre el archivo CSV de solicitudes y procesa cada línea de manera secuencial. Para cada solicitud, extrae el nombre de la familia, la hora solicitada y la cantidad de personas; con esta información construye una estructura Peticion que envía al Controlador. Luego espera la respuesta correspondiente, la imprime en pantalla y prosigue con la siguiente solicitud 2 segundos después. Cuando el archivo finaliza, la función se encarga de cerrar los pipes utilizados, eliminar el pipe creado por el Agente y liberar los recursos utilizados.

Función main()

Integra y coordina todas las operaciones del Agente. Al iniciar, abre un semáforo nominal que será utilizado por el Controlador para indicar el fin de la simulación. Luego invoca a tomarArgumentosAgente(), luego revisa si la validación de argumento es exitosa y procede a ejecutar la función leerArchivo(). Una vez procesadas todas las solicitudes, el Agente queda bloqueado en espera de la señal del Controlador. Cuando el Controlador activa el semáforo para finalizar la ejecución de todos los Agentes, este proceso imprime un mensaje de terminación y libera la memoria utilizada.

Controlador

El Controlador de Reservas constituye el principal elemento del simulador, actuando como servidor dentro del modelo cliente-servidor. Este proceso central administra las solicitudes enviadas por los Agentes de Reserva, mantiene la información del estado del parque para cada hora de la simulación y ejecuta la lógica de asignación, reprogramación o negación de reservas, de acuerdo con las restricciones establecidas en el enunciado del proyecto. Su funcionamiento se estructura alrededor de un conjunto de funciones que permiten la administración de argumentos, inicialización de los parques, gestión del reloj del simulado, recepción de peticiones y generación de reportes tanto periódicos (cada hora) como finales. El Controlador también emplea hilos POSIX para dividir su funcionamiento en tres procesos concurrentes.

Función manipularReloj()

Esta función implementa la simulación del avance del tiempo. Esta función actualiza la hora actual, y utiliza un mecanismo de condición para notificar a otros hilos que una nueva hora ha transcurrido. De esta manera, la simulación del tiempo avanza de forma controlada y sincronizada, permitiendo que el resto de los subsistemas reaccionen oportunamente a cada cambio horario.

Función tomarArgumentosControlador()

Esta función se encarga de validar los parámetros proporcionados desde la línea de comandos al ejecutar el controlador, tales como la hora inicial, la hora final, la duración en segundos de cada hora simulada, el aforo máximo permitido y el nombre del named pipe principal por donde se recibirán las peticiones de los agentes. La función verifica la presencia de todas las banderas, comprueba que los valores estén dentro de los rangos establecidos (por ejemplo, que las horas estén entre 7 y 19), y finalmente almacena toda la información en una estructura RetornoArgumentos. En caso de valores inválidos o ausencia de parámetros obligatorios, esta función retorna un estado de error, impidiendo que el sistema continúe.

Función inicializarParques()

Esta función crea y configura la estructura de datos utilizada para representar el estado del parque durante cada hora de la simulación. Para cada hora entre la hora inicial y la hora final, se inicializan contadores de familias, contadores de entradas y salidas, número de personas en el parque, y se asigna memoria para almacenar información de cada familia. Este arreglo de parques constituye la base de datos que consultará el Controlador para determinar si una reserva puede aceptarse, reprogramarse o negarse ya que cada uno representa el parque en una hora distinta del día.

Función reportePorHora()
Esta función permanece en espera de una señal enviada por el reloj y cuando la recibe, calcula e imprime las familias que ingresan o salen en esa hora, así como la cantidad de personas involucradas. Este componente permite generar un seguimiento en tiempo real del comportamiento del parque.

Función recibirMensajes()
Esta función es encargada de recibir todas las peticiones enviadas por los Agentes de Reserva a través del named pipe principal enviado como parámetro tanto al controlador como a los Agentes. Esta función analiza cada solicitud, revisa la disponibilidad del parque en las horas correspondientes y determina si la reserva puede ser aprobada, reprogramada para una hora posterior o negada según las restricciones de aforo y disponibilidad de dos horas consecutivas. La función actualiza la estructura de parques de acuerdo con cada decisión y envía la respuesta correspondiente al pipe del agente que haya realizado la solicitud. Adicionalmente, registra la conexión inicial de cada agente, responde con la hora actual del sistema y contabiliza la cantidad de agentes activos, dato utilizado posteriormente para finalizar su ejecución mediante un named semaphore.

Función reporteFinal()
Se ejecuta después de que concluye la simulación. En ella se recorren las estructuras de datos creadas durante el día simulado para determinar las horas de mayor y menor aforo, el número total de solicitudes aceptadas, negadas, reprogramadas, y finalmente se envía un mensaje especial por el named pipe principal para indicar al hilo de recibirMensajes() que no habrá más solicitudes y que el día ya ha acabado. Esta función produce el informe final que sintetiza el comportamiento del parque durante toda la jornada simulada.

Función main()
Coordina todas las operaciones anteriores. Se encarga de abrir el named semaphore que permitirá finalizar posteriormente a los agentes, inicializar el reloj, inicializar las estructuras de parques, crear los hilos necesarios para el manejo del tiempo, el reporte cada hora y la recepción de mensajes. Una vez finaliza la simulación del día, genera el reporte final y utiliza la función sem_post() la misma cantidad de veces que agentes hayan sido registrados para despertarlos a todos, dando así por concluida la ejecución del sistema de forma ordenada y sincronizada.

Estructura del proyecto y Makefile

La estructura del proyecto se organiza de manera modular. En la raíz del proyecto se encuentran los archivos principales de cada proceso del sistema, es decir los ficheros AgenteDeReservas.c y ControladorDeReservas.c, los cuales contienen las funciones main que dan inicio a la ejecución del Agente y del Controlador, respectivamente. De igual manera, en este mismo nivel se encuentra el archivo Makefile que automatiza la compilación del sistema. El proyecto está dividido en dos directorios principales que agrupan los módulos que conforman la lógica del sistema. El primero, ModulosDeDefinicion, almacena los archivos header ModuloAgente.h y ModuloControlador.h. Estos archivos contienen las declaraciones de estructuras de datos, variables globales, prototipos de funciones y todos aquellos elementos que deben ser compartidos entre los distintos archivos fuente. El segundo directorio, ModulosDeImplementacion, contiene los archivos ModuloAgente.c y ModuloControlador.c, cada uno encargado de implementar las funciones declaradas en los archivos header. Esta separación permite mantener el código organizado siguiendo principios de modularidad, ya que la lógica operativa del agente y del controlador se encuentra agrupada y aislada del resto de la estructura del proyecto. Gracias a esta división, tanto el AgenteDeReservas.c como el ControladorDeReservas.c pueden incluir únicamente los encabezados necesarios y delegar su funcionalidad a los módulos correspondientes. Adicionalmente, el directorio Ejecutables se utiliza como destino de compilación. Los archivos objeto generados durante el proceso de construcción, así como los ejecutables finales de cada componente, se almacenan allí. Este estilo asegura que el producto de la compilación no se mezcle con el código fuente. El archivo Makefile presente en la raíz del proyecto coordina todo el proceso de compilación. Este automatiza la construcción de los ejecutables AgenteDeReservas y ControladorDeReservas compilando primero los módulos de implementación y luego enlazándolos con los archivos que contienen las funciones main. El Makefile crea el directorio de ejecutables cuando es necesario, administra la compilación modular mediante archivos objeto y proporciona una regla clean que elimina todo el contenido generado, permitiendo así recompilaciones completas cuando se requiera.

Plan de Pruebas

Introducción

El propósito de este plan de pruebas es validar el correcto funcionamiento del sistema de reservas del parque, compuesto por el **Controlador** y los **Agentes**, asegurando que cumpla con los requisitos funcionales y no funcionales definidos en el proyecto.
El plan establece los objetivos, procedimientos, casos de prueba, criterios de aceptación y estructura de evidencias necesarias para demostrar el correcto comportamiento bajo condiciones normales, límites, errores y concurrencia.

Alcance de las pruebas

Este plan cubre:

- Validación de parámetros de entrada del Controlador y del Agente.
- Verificación de comportamiento con parámetros en órdenes distintos.
- Manejo de errores en ejecución (parámetros faltantes, valores inválidos, archivos inexistentes, pipes no accesibles).
- Procesamiento correcto de solicitudes de reserva según reglas del sistema.
- Respuestas del Controlador ante solicitudes válidas, inválidas, extemporáneas o que exceden el aforo.
- Pruebas de concurrencia con múltiples agentes simultáneos.
- Generación y validación del reporte final del sistema.

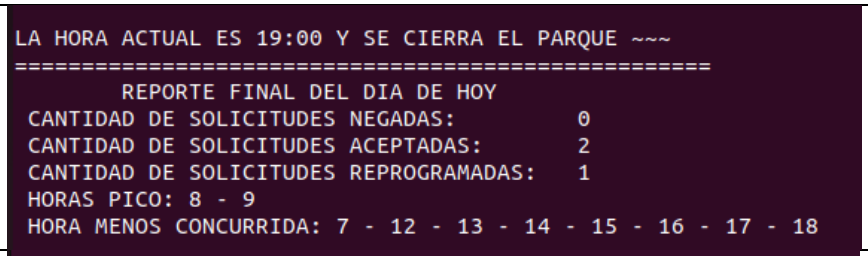
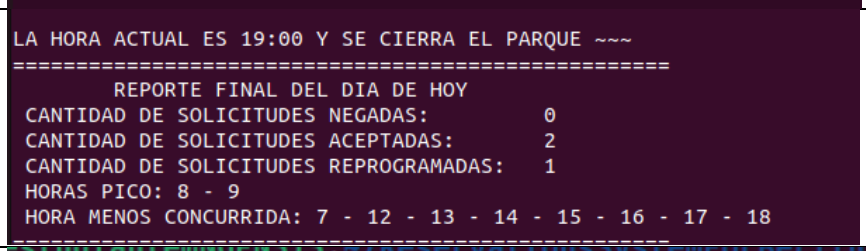
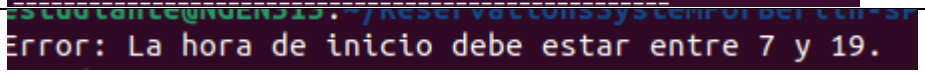
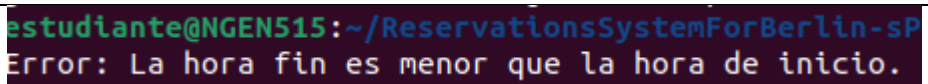
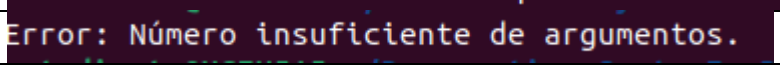
Casos de prueba

ID	Objetivo	Entrada / comando	Resultado esperado
C1	Inicio correcto del Controlador	./Ejecutables/ControladorDeReservas -i 7 -f 19 -s 2 -t 10 -p prueba	Inicia simulación y crea pipe
C2	Flags en orden aleatorio	./Ejecutables/ControladorDeReservas -s 2 -p prueba -t 10 -f 19 -i 7	Igual ejecución que C1
C3	Error por horaIni fuera de rango	./Ejecutables/ControladorDeReservas -i 6 -f 19 -s 2 -t 10 -p pruebac	Mensaje de error y salida
C4	Error horaIni > horaFin	./Ejecutables/ControladorDeReservas -i 18 -f 10 -s 2 -t 10 -p prueba	Mensaje de error
C5	Falta parámetro obligatorio (-p)	./Ejecutables/ControladorDeReservas -i 7 -f 19 -s 2 -t 10	Error por parámetro faltante
C6	segHoras inválido (0 o negativo)	./Ejecutables/ControladorDeReservas -i 7 -f 19 -s 0 -t 10 -p prueba	Error por valor inválido
C7	Ejecución	./Ejecutables/AgenteDeReservas -s A1 -a	Envía solicitudes y recibe

	correcta del Agente	prueba -p prueba	respuestas
C8	Flags del Agente en orden aleatorio	./Ejecutables/AgenteDeReservas -p prueba -s A1 -a prueba	Igual que C7
C9	Archivo de solicitudes inexistente	./Ejecutables/AgenteDeReservas -s A1 -a prueba -p noHay	Error: archivo no encontrado
C10	CSV malformado	Archivo con líneas incompletas	Mensaje de error por línea
C11	Solicitud extemporánea	Solicitar hora < hora actual	reprogramación
C12	Solicitud que excede aforo	Ej: cantidad 11 con aforo 10	Rechazo con mensaje
C13	Concurrencia con varios agentes	3 agentes simultáneos	Asignaciones correctas sin race conditions

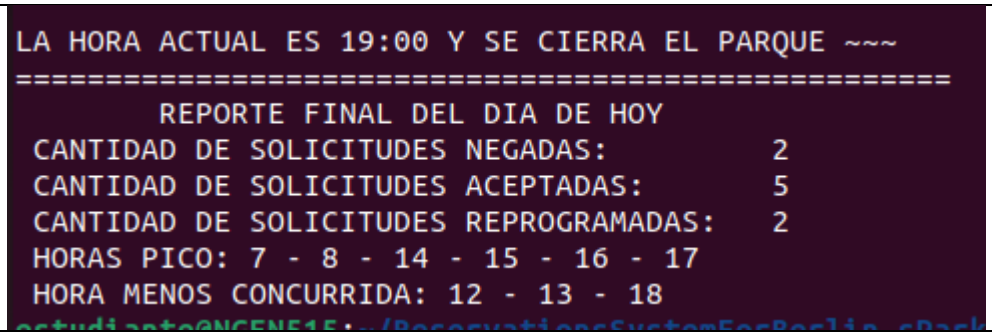
Resultados de las pruebas

A continuación, se documentan los resultados obtenidos tras ejecutar los casos definidos en la tabla anterior

ID	Resultado esperado	Evidencia
C1	Inicia correctamente	
C2	Acepta flags en cualquier orden	
sC3	Error por hora fuera de rango	
C4	Error horaIni >horaFin	
C5	Error	

	parámetro faltante	
C6	segHoras inválido (0 o negativo)	estudiante@NGEN515:~/ segHoras es inválido
C7	Ejecución correcta del Agente	UN AGENTE SE REGISTRÓ, SE LLAMA: A1 HORA ACTUAL RECIBIDA DESDE EL CONTROLADOR 7 PETICION: { NombreFamilia: aponte HoraSolicitada: 8:00 PersonasSolicitadas: 10 } RESPUESTA: RESERVA OK: TODO BIEN PETICION: { NombreFamilia: tobar HoraSolicitada: 9:00 PersonasSolicitadas: 2 } RESPUESTA: RESERVA REPROGRAMADA: PARA EL MISMO DIA A LAS 10:00 PETICION: { NombreFamilia: ramirez HoraSolicitada: 10:00 PersonasSolicitadas: 3 } RESPUESTA: RESERVA OK: TODO BIEN AGENTE A1 TERMINA.

C8	Flags del Agente en orden aleatorio	s A1 -a prueba HORA ACTUAL RECIBIDA DESDE EL CONTROLADOR 7 PETICION: { NombreFamilia: aponte HoraSolicitada: 8:00 PersonasSolicitadas: 10 } RESPUESTA: RESERVA OK: TODO BIEN PETICION: { NombreFamilia: tobar HoraSolicitada: 9:00 PersonasSolicitadas: 2 } RESPUESTA: RESERVA REPROGRAMADA: PARA EL MISMO DIA A LAS 10:00 PETICION: { NombreFamilia: ramirez HoraSolicitada: 10:00 PersonasSolicitadas: 3 } RESPUESTA: RESERVA OK: TODO BIEN AGENTE A1 TERMINA.
C9	Archivo de solicitudes inexistente	rror al abrir el archivo de solicitudes: No such file or directory
C10	CSV malformado	HORA ACTUAL RECIBIDA DESDE EL CONTROLADOR 7 Error: línea CSV inválida Error: línea CSV inválida
C11	Solicitud extemporánea	HORA ACTUAL RECIBIDA DESDE EL CONTROLADOR 7 PETICION: { NombreFamilia: aponte HoraSolicitada: 6:00 PersonasSolicitadas: 10 } RESPUESTA: RESERVA REPROGRAMADA: PARA EL MISMO DIA A LAS 7:00
C12	Solicitud que excede aforo	HORA ACTUAL RECIBIDA DESDE EL CONTROLADOR 7 PETICION: { NombreFamilia: aponte HoraSolicitada: 6:00 PersonasSolicitadas: 11 } RESPUESTA: RESERVA NEGADA: SU FAMILIA EXCEDE EL AFORO MÁXIMO DEL PARQUE
C13	Concurrencia con varios agentes	UN AGENTE SE REGISTRÓ, SE LLAMA: A1 UN AGENTE SE REGISTRÓ, SE LLAMA: A2 UN AGENTE SE REGISTRÓ, SE LLAMA: A3 ==========

Conclusiones

Tras la ejecución de las pruebas definidas, se determinó si el sistema cumple adecuadamente los requisitos funcionales y de robustez.
La información registrada en la sección de resultados permite verificar:

- El comportamiento correcto del Controlador al validar parámetros, procesar solicitudes y generar el reporte final.
- La correcta interacción cliente–servidor mediante pipes con múltiples agentes simultáneos.
- La adecuada detección de errores, parámetros inválidos, archivos malformados y condiciones de estrés.

Conclusiones Generales

- El sistema cumple su propósito al simular un proceso de reservas concurrente, gestionando correctamente aforo, solicitudes y reprogramaciones mediante pipes y sincronización POSIX.
- La arquitectura cliente-servidor y la división del Controlador en hilos permiten un funcionamiento estable, ordenado y libre de condiciones de carrera.
- Las pruebas realizadas confirman que el sistema responde adecuadamente tanto a entradas válidas como a errores, manteniendo robustez y coherencia en escenarios normales y concurrentes.
- La organización modular y el uso de un Makefile facilitan la comprensión, mantenimiento y ejecución del proyecto.

Bibliografía

AI-FutureSchool. (2025, enero 4). *Mutex en programación: Conceptos y mejores prácticas*. AI-FutureSchool. https://www.ai-futureschool.com/es/informatica/mutex-sincronizacion-en-programacion.php

AIX. (2025, julio 11). https://www.ibm.com/docs/es/aix/7.3.0?topic=p-pthread-mutex-lock-pthread-mutex-trylock-pthread-mutex-unlock-subroutine

close(2)—Linux manual page. (s. f.). Recuperado 18 de noviembre de 2025, de https://man7.org/linux/man-pages/man2/close.2.html?utm_source=chatgpt.com

fgets(3p)—Linux manual page. (s. f.). Recuperado 18 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/fgets.3p.html?utm_source=chatgpt.com

fopen(3)—Linux manual page. (s. f.). Recuperado 18 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/fopen.3.html?utm_source=chatgpt.com

malloc(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/free.3.html

Manejo de Punteros y Memoria Dinámica | Sistemas Operativos—UTN FRBA. (s. f.). Recuperado 13 de noviembre de 2025, de https://docs.utnso.com.ar/guias/programacion/punteros

mkfifo(3)—Linux manual page. (s. f.). Recuperado 18 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/mkfifo.3.html?utm_source=chatgpt.com

open(2)—Linux manual page. (s. f.). Recuperado 18 de noviembre de 2025, de https://man7.org/linux/man-pages/man2/open.2.html?utm_source=chatgpt.com

Pthread_cond_signal. (s. f.). Recuperado 13 de noviembre de 2025, de https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_cond_signal.html

Pthread_cond_wait. (s. f.). Recuperado 13 de noviembre de 2025, de https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_cond_wait.html

pthread_create(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/pthread_create.3.html

pthread_join(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/pthread_join.3.html

read(2)—Linux manual page. (s. f.). Recuperado 18 de noviembre de 2025, de https://man7.org/linux/man-pages/man2/read.2.html?utm_source=chatgpt.com

sem_close(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/sem_close.3.html

sem_open(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/sem_open.3.html

sem_overview(7)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man7/sem_overview.7.html

sem_post(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/sem_post.3.html

sem_unlink(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/sem_unlink.3.html

sem_wait(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/sem_wait.3.html

shm_overview(7)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man7/shm_overview.7.html

Silberschatz, A., Galvin, P., & Gagne, G. (2020). *Operating System Concepts* (Tenth). Wiley.

sleep(3)—Linux manual page. (s. f.). Recuperado 18 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/sleep.3.html

unlink(2)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man2/unlink.2.html

write(2)—Linux manual page. (s. f.). Recuperado 18 de noviembre de 2025, de https://man7.org/linux/man-pages/man2/write.2.html?utm_source=chatgpt.com

Z/OS. (2021, junio 25). https://www.ibm.com/docs/en/zos/2.4.0?topic=functions-pthread-cond-wait-wait-condition-variable

Z/OS. (2023, abril 28). https://www.ibm.com/docs/en/zos/2.5.0?topic=functions-pthread-mutex-unlock-unlock-mutex-object