Trabajo Práctico N°1
Sistemas Operativos (72.08)

Integrantes:

- Nicole Belén Bertucci - Legajo 64415
- Juan Ignacio Bridoux - Legajo 64092
- Sebastián Nicolás Lee - Legajo 64675

Profesores:

- Ariel Godio
- Alejo Ezequiel Aquili
- Gonzalo Manuel Beade
- Fernando Gleiser Flores
- Guido Matías Mogni

Fecha de entrega: 14/09/2025

Índice:

1. Introducción:

En el presente trabajo se desarrolla una implementación del juego ChompChamps, en el marco de la materia Sistemas Operativos, haciendo foco en la utilización de diferentes tipos de IPCs de sistemas POSIX. El juego ChompChamps se desarrolla sobre un tablero rectangular compuesto por celdas que contienen valores numéricos. Cada jugador comienza en una posición inicial predeterminada y, en su turno, puede desplazarse a una celda adyacente. Al hacerlo, suma a su puntaje el valor de esa celda, que luego queda marcada como ocupada. A esto se le agrega que ChompChamps no es un juego por turnos y permite que un jugador pueda jugar varios movimientos si es que los demás jugadores quedan “pensantes”. La partida continúa hasta que todos los jugadores quedan bloqueados, es decir, sin movimientos válidos disponibles alrededor de su posición. El ganador se determina como el jugador con mayor puntaje total acumulado. En caso de empate en el puntaje, se utilizan criterios de desempate: primero se considera la cantidad de jugadas válidas realizadas y, de persistir la igualdad, el jugador con menor cantidad de jugadas inválidas es declarado ganador.

El proyecto consiste en tres programas que interactúan de manera coordinada:

- master.c, que administra el flujo del juego, mantiene el estado global y gestiona la comunicación con los demás procesos
- <jugadores>.c, que actúan como procesos independientes que calculan y envían movimientos por pipes
- view.c, que se encarga de representar gráficamente el tablero y mostrar la evolución del juego en tiempo real

Para lograr esta interacción se emplean memoria compartida (para el estado global), pipes (para transmitir movimientos de los jugadores al máster) y semáforos POSIX (para garantizar la correcta sincronización entre procesos). De este modo, el trabajo permite ejercitar el uso combinado de distintos mecanismos de IPC (Inter-Process Communication) y resolver los problemas de coordinación entre múltiples procesos lectores y escritores.

En este informe, se documenta brevemente el diseño, implementación y funcionamiento de cada componente, así como las decisiones tomadas para la distribución inicial de los jugadores, el manejo de turnos y la presentación visual de la partida.

2. Archivos fuente:

El proyecto se compone de distintos archivos fuente en C, cada uno con una responsabilidad dentro del sistema. Esta división modular permite separar claramente la lógica de control, la visualización y las estrategias, garantizando un diseño organizado y extensible. A continuación se detallan por separado estos archivos.

2.1. master.c

Este archivo implementa el proceso principal del juego, encargado de coordinar la interacción entre todos los demás. El máster inicializa los parámetros de la partida, crea las memorias compartidas y semáforos, lanza a los jugadores y al proceso de vista, y controla el ciclo de ejecución del juego. Además, se encarga de recibir las jugadas a través de los pipes y validar cada movimiento, actualizando el estado global de manera sincronizada.

2.2. view.c

La vista es el proceso que se encarga de representar gráficamente el tablero y el puntaje de los jugadores en tiempo real. Utiliza memoria compartida para leer el estado del juego y semáforos para sincronizar su ejecución con el máster, evitando lecturas inconsistentes. El archivo contiene múltiples funciones para imprimir el tablero con colores, mostrar el leaderboard y finalmente destacar a los tres primeros puestos en un podio. Aunque no influye en la lógica del juego, resulta clave para la visualización clara y estética de la dinámica de la partida. En este caso, view.c está inspirado en un estanque donde viven los diferentes jugadores (patitos), los cuales compiten por obtener las zonas en las que mayor frecuencia reciben comida (representado por los puntos), por esto mismo se eligen los colores azul y verde para el tablero y el fondo. Asimismo cada uno de los jugadores esta distinguido por un emoticón de pato (🦆) y un color significativo.

2.3. Jugadores

Los jugadores son procesos independientes creados por el máster, que se comunican con él a través de pipes y acceden al estado del tablero mediante memoria compartida. Cada turno, esperan la señal de sincronización desde el máster (mediante semáforos), leen de manera controlada el estado global y calculan su próxima jugada. Luego, escriben ese movimiento en el pipe asociado para que el máster lo procese. Aunque todos siguen este mismo flujo de ejecución, la diferencia entre ellos radica en la estrategia utilizada para decidir el movimiento, lo que permite comparar distintos estilos de juego.

2.3.1. quackenstein.c

Este es un jugador que tuvo un comportamiento inesperado cuando se lo probó, pero como sus resultados se podían considerar aceptables, decidimos mantenerlo como un jugador extra.

2.3.2. bonaquackte.c

Su estrategia es más sofisticada, ya que implementa una búsqueda recursiva limitada en profundidad. Esto le permite proyectar puntos potenciales a futuro y tomar decisiones más informadas, priorizando caminos que optimicen la suma de beneficios en varios movimientos.

2.3.3. randomQuacks.c

Este jugador sigue un enfoque aleatorio. Cada turno elige un movimiento de manera pseudoaleatoria, sin analizar el tablero ni planificar. Aunque poco competitivo, es útil para probar el sistema y aportar variabilidad a las partidas.

2.4 /libs

Los archivos contienen la implementación funcional del sistema desde la lógica de juego y la inicialización de procesos, hasta la lectura de parámetros y la gestión de memoria y sincronización. Cada uno cumple un rol modular específico, lo que permite mantener el código organizado y separar claramente responsabilidades como control de estado, comunicación o entrada de parámetros.

2.4.1. game.c

Define la lógica central del tablero y los movimientos. Aquí se implementan funciones para ubicar a los jugadores al inicio, validar jugadas, esperar movimientos a través de los pipes y determinar cuándo un jugador o todos quedan bloqueados. Actúa como el núcleo de la dinámica de juego.

2.4.2. init.c

Se encarga de inicializar tanto el estado del juego como las estructuras de sincronización. Contiene la creación y destrucción de semáforos, la inicialización de los pipes, el lanzamiento de procesos jugadores y vista, y la configuración del tablero inicial con valores pseudoaleatorios.

2.4.3. parameters.c

Gestiona la lectura y validación de los parámetros ingresados por consola (opciones como tamaño del tablero, cantidad de jugadores, delay, timeout, etc.). Establece valores por defecto y verifica restricciones, garantizando que la ejecución comience con datos consistentes.

2.4.4. shm.c

Implementa funciones auxiliares para manejar la memoria compartida. Incluye la creación, mapeo, cierre y eliminación de segmentos de memoria, que sirven como medio de comunicación entre máster, jugadores y vista.

2.4.5. sync.c

Contiene la lógica de sincronización mediante semáforos POSIX. Define las funciones para coordinar el acceso entre máster y jugadores (lectores-escritor), así como las señales de turnos y de impresión hacia la vista. Es fundamental para evitar condiciones de carrera.

2.5 /include

Los archivos agrupan las definiciones necesarias para compartir estructuras, constantes y prototipos de funciones entre los distintos módulos. En particular, structs.h establece la representación de jugadores, tablero y sincronización, mientras que los demás headers garantizan que cada archivo fuente pueda interactuar con el resto sin depender directamente de su implementación.

3. Instrucciones de compilación y ejecución:

Para poder comenzar es necesario inicializar el contenedor docker (teniendo el contenedor de la cátedra ya instalada) con:

```
./docker.sh
```

Luego moverse al root con:

```
cd root
```

Para compilar el proyecto, usar el comando:

```
make all
```

Para ejecutarlo, escribir el comando ./bin/master con los parámetros:

- -p player1 player2 … : Ruta/s de los binarios de los jugadores, como mínimo un jugador y como máximo nueve jugadores
- -w width: Ancho del tablero, default y mínimo de 10 (Opcional)
- -h height: Alto del tablero, default y mínimo de 10 (Opcional)
- -d delay: Tiempo que espera el master cada vez que se imprime el estado, en milisegundos. Default 200ms (Opcional)

- -t timeout: TImeout en segundos para recibir solicitudes de movimientos válidos. Default 10s (Opcional)
- -s seed: Semilla utilizada para la generación del tablero. Default time(NULL) (Opcional)
- -v view: Ruta del binario de la vista. Default sin vista (Opcional)

4. Rutas relativas de la vista y jugador para el torneo:

Luego de la compilación se podrán utilizar dentro de los parámetros los binarios compilados por el Makefile de algunos jugadores y de la vista:

Se compila como posibles jugadores:
- ./bin/quackenstein
- ./bin/bonaquackte
- ./bin/randomQuacks

Se compila como posible vista:
- ./bin/view

5. Problemas y limitaciones durante el desarrollo y decisiones tomadas:

Listamos a continuación una serie de los problemas con los que nos topamos a lo largo del proyecto, junto con su solución o decisión tomada al respecto.

Posicionamiento de jugadores: Con el fin de crear una experiencia justa para todos los jugadores se implementó una división del tablero equitativa en forma rectangular y luego el posicionamiento de estos dentro de su sección tal que cada uno tuviese aproximadamente la mismas chances de encontrarse bloqueado.

Flickering: Encontramos que existe un problema de flickering con la vista en el caso de correr el programa con un delay bajo (notable hasta -d 100). Esto se debía al gran número de prints que se utilizan en la vista que realizamos. Aún así, aprovechamos el hecho de que las zonas en las que se imprime son fijas, por lo que en lugar de mover el cursor y limpiar la terminal, simplemente movemos el cursor para sobreescribir en el tablero antiguo. Con esta corrección, el flickering ya no es visible (salvo por el cursor).

Uso de qsort en view: Al incluir en nuestro view un leaderboard en tiempo real, fue necesario utilizar la función qsort para ordenar a los jugadores según el criterio de posiciones, lo que significa invertir tiempo para dicha acción. Aún así, decidimos implementarlo ya que lo consideramos una mejora visual significativa y que el tiempo que se pierde en ordenar a los jugadores es escaso al ser un arreglo de pocos elementos.

Límite de cifras: El leaderboard previamente mencionado está implementado para imprimir hasta un máximo de tres cifras para cada columna (puntos, movimientos válidos y movimientos inválidos) y, de exceder 999, el “leaderboard” imprime desfasado. Inicialmente, al implementarlo, no tuvimos en

cuenta el hacer los respectivos print del “leaderboard” dinámicamente contemplando cantidades de cifras diferentes. Nos dimos cuenta de este problema al momento de testear varios parámetros. Se decidió dejarlo con este máximo de tres decimales hardcodeados primeramente porque esperamos, aunque sea posible, que no se jueguen partidas tan extensas en donde se supere el valor de 999 en cualquiera de las tres estadísticas.

Límite de ancho de pantalla: La vista presenta problemas con valores de ancho de tablero altos. El problema preciso es que decidimos imprimir sobre los laterales unas columnas que simulan ser pasto alrededor del agua (que sería el tablero), y dicho espacio podría aprovecharse para generar tableros más extensos. Aún así, consideramos que esa distribución de espacios era agradable para la vista, y en caso de necesitar más espacio se puede reducir el ancho del diseño alrededor del tablero.

Acceso a información útil: Notamos que muchas veces en las que intentábamos avanzar con el proyecto nos estancábamos al no estar seguros de cómo hacer uso de ciertas herramientas y evitar errores en su uso. Sin embargo, muchas de estas dudas se fueron resolviendo con el correr de las clases, en las que se veían contenidos útiles para el TP.

6. Citas de fragmentos de código utilizado de otras fuentes:

Para cualquier duda que hubiese surgido durante el proceso del proyecto se utilizó principalmente el comando “man”. También, encontramos el canal de YouTube “Jacob Sorber” que nos dio un buen introductorio teórico de varios sistemas de POSIX. Por último, destacamos el ejercicio 3 de la teórica de semáforos que se resolvió durante la clase teórica virtual, que resultó de gran utilidad para el uso de los MUTEXs del TP.