# Trabajo Práctico 1: ChompChamps

**Integrantes:**

Pessagno Bautista 65101

Perrone Rocco 65628

Ordano Pascal 65485

## Resumen del Proyecto

ChompChamps es un juego multijugador implementado en C que simula un juego de estrategia donde múltiples jugadores compiten en un tablero bidimensional. El proyecto utiliza técnicas avanzadas de programación de sistemas, incluyendo memoria compartida, semáforos POSIX, pipes y comunicación entre procesos para coordinar el juego en tiempo real.

## Decisiones Tomadas Durante el Desarrollo

Durante el desarrollo del proyecto ChompChamps, se tomaron varias decisiones arquitectónicas fundamentales que determinaron la estructura y funcionamiento del sistema. La primera decisión importante fue implementar una arquitectura maestro-esclavo con procesos separados para cada componente. Esta elección permite modularidad, escalabilidad y facilita la depuración, ya que el proceso maestro coordina el juego mientras que cada jugador y la vista son procesos independientes que pueden ser desarrollados y probados por separado.

Para el manejo de la comunicación no bloqueante con los procesos jugadores se decidió utilizar la función poll(). Esta función permite al maestro monitorear múltiples descriptores de archivo (pipes de comunicación con cada jugador) de manera simultánea, sin quedar bloqueado esperando la respuesta de un único jugador. Gracias a poll(), el maestro puede detectar de forma eficiente qué jugador envió datos y reaccionar en consecuencia, mejorando la responsividad y evitando cuelgues por procesos inactivos.

Además, para la asignación de turnos de los jugadores se adoptó la política de planificación Round Robin. Este esquema garantiza que todos los jugadores participen de manera justa y ordenada: El master tiene una variable que registra el último jugador el cual se le procesó un movimiento. al comenzar el siguiente ciclo del juego, la ronda va a iniciar por el jugador siguiente al que se proceso la última jugada, recorriendo los pedidos de forma circular, de esta forma permitiendo que todos los jugadores tengan una cantidad similar de movimientos. Cuando llega al último jugador, la variable vuelve a cero, reiniciando el ciclo. Esta estrategia evita la inanición, distribuye equitativamente las oportunidades de juego y simplifica la lógica de control de turnos.

Para el sistema de comunicación entre procesos, se decidió utilizar memoria compartida para el estado del juego y semáforos POSIX para la sincronización. La memoria compartida permite acceso eficiente al estado del juego por parte de todos los procesos, mientras que los semáforos garantizan la sincronización correcta y evitan condiciones de carrera. Esta combinación proporciona un balance óptimo entre rendimiento y seguridad.

El uso de poll() complementa este diseño al facilitar la lectura asíncrona de pipes, integrando la comunicación maestro-jugadores dentro de un bucle centralizado de eventos.

## Estrategia de Sincronización

La estrategia de sincronización implementada utiliza un sistema de semáforos POSIX nombrados, definidos en game_semaphore.h, que permiten un control granular del acceso concurrente al estado del juego. Los principales son:

- /game_A (view_update_signal): utilizado por el proceso maestro para notificar a la vista que hay cambios en el estado del juego que deben ser dibujados.

- /game_B (view_draw_complete): utilizado por la vista para informar al maestro que terminó de actualizar la pantalla.

- /game_C (master_access_mutex): mutex que evita la inanición del maestro cuando accede al estado del juego.

- /game_D (game_state_mutex): mutex principal que protege la sección crítica del estado del juego en memoria compartida.

- /game_E (reader_count_mutex): mutex que controla de manera segura la variable reader_count, utilizada para contar los procesos que están leyendo el estado del juego.

- /game_G_i (player_turn_sem[i]): conjunto de semáforos individuales (uno por jugador) que controlan los turnos de cada participante.

Este diseño permite coordinar eficientemente las interacciones entre el maestro, la vista y los jugadores, evitando race conditions y garantizando la integridad del estado compartido.

Además de los semáforos, el maestro utiliza poll() como mecanismo auxiliar de sincronización con los jugadores a través de pipes no bloqueantes. Mientras los semáforos controlan el acceso a la memoria compartida y la correcta secuencia de turnos, poll() asegura que el maestro no quede detenido esperando indefinidamente la respuesta de un proceso.

## Instrucciones de Compilación y Ejecución

Para compilar y ejecutar el proyecto ChompChamps, es necesario cumplir con ciertos prerrequisitos del sistema. El proyecto requiere un compilador C compatible con el estándar C99, como gcc o clang, la biblioteca ncurses para la interfaz gráfica, y un sistema operativo POSIX como Linux o macOS.

La compilación del proyecto se realiza mediante el sistema de construcción Make. El comando make all compila todos los ejecutables, incluyendo el proceso maestro, la interfaz gráfica y los jugadores. También es posible compilar componentes específicos con make master, make view, make player_competitive, make player_greedy o make player_random.

La ejecución del juego se realiza a través del proceso maestro. El modo básico es ./master 10 10, que utiliza jugadores por defecto en un tablero de 10x10. También puede ejecutarse con ./master 15 15 ./player_competitive ./player_greedy ./player_random para un tablero de 15x15 con tres jugadores distintos. El formato general es ./master <ancho> <alto> [jugadores...], con un rango de entre 5 y 50 celdas y hasta 9 jugadores simultáneos.

## Rutas Relativas para el Torneo

El proceso de visualización se encuentra en ./view y muestra el tablero y las estadísticas en tiempo real.

## Limitaciones

- Tamaño máximo del tablero: 50x50.
- Máximo de 9 jugadores simultáneos.
- Dependencia de ncurses y terminal compatible.
- Solo sistemas POSIX.

Rendimiento:

- Pipes con latencia.
- Uso de memoria proporcional al tablero.
- Semáforos que pueden crear cuellos de botella.

Escalabilidad:

- Un proceso por jugador.
- La complejidad de sincronización crece con más jugadores.

Problemas Encontrados y Soluciones

- Deadlocks: resueltos con timeouts e inactividad.
- Race conditions: resueltas con /game_C y /game_D.
- Compatibilidad macOS: semáforos nombrados y flags específicos.
- Pipes bloqueantes: reemplazados por poll() en pipes no bloqueantes.
- Jugadores bloqueados: se implementó la función has_valid_move() para detección temprana.

Citas de Código Reutilizado

El código es original, basado en patrones estándar:

- Semáforos POSIX nombrados (/game_A, /game_B, /game_C, /game_D, /game_E, /game_G_i).

- BFS clásico.

- Memoria compartida con shm_open() y mmap().

- Interfaz gráfica con ncurses.