72.11 Sistemas Operativos

Trabajo Práctico 1: ChompChamps

Integrantes:

- Franco Branda - 65506
- Manuel García Puente - 65505
- Mateo Agustin Arias - 65613

Fecha de Entrega: 14/09/2025

Grupo: 12

2025

Objetivo

El objetivo del presente trabajo consistió en implementar un sistema concurrente donde un proceso **maestro** coordina, a partir de parámetros de ejecución, procesos **jugador** que envían movimientos y un proceso **vista** que muestra el tablero en tiempo real, utilizando *pipes*, memoria compartida, semáforos y select para garantizar sincronización, justicia (round-robin) y finalización por *timeout*.

Proceso Player

Decisiones tomadas

- **Rol y protocolo:** cada jugador la espera habilitación (semáforo player_can_move[i]), toma un snapshot seguro del estado bajo el protocolo lectores–escritor y calcula un movimiento local sin bloquear al master.
- **Estrategia de movimiento:** se implementaron 2 estrategias principales, una para cuando hay un solo jugador donde el player recorre todo el tablero haciendo una forma de espiral y así conseguir todos los puntos. Para cuando hay 2 o más jugadores, se eligió una heurística simple que consiste en un algoritmo greedy donde siempre se dirige a la casilla disponible con mayor puntaje. Ambos cuentan con un número máximo de intentos por turno para impedir bucles infinitos si el tablero está casi agotado.
- **Comunicación:** el movimiento se envía por su pipe dedicado. La validación final siempre la hace el master.
- **Limpieza:** cada jugador desmonta sus mapeos (munmap) y sale al recibir señal o al finalizar el juego.

Limitaciones

- **Inteligencia limitada:** la heurística no es óptima ni adaptable; no aprende del estado pasado ni coordina con otros jugadores.
- **Conocimiento parcial:** el jugador decide con snapshot; podrían existir ligeras carreras semánticas frente a cambios muy rápidos (que se resuelven del lado del master).
- **Reutilización:** la lógica está acoplada a la estructura de game_state; cambios de reglas/tablero requieren recompilar.

Proceso Maestro

Decisiones tomadas:

- **Arquitectura e IPC:** el master orquesta el juego y centraliza la lógica de validación/actualización del estado. Se separó la memoria compartida en dos regiones: (1) game_state (tablero, puntuaciones y metadatos) y (2) game_sync (semaforización y contadores), para desacoplar datos del mecanismo de sincronización y simplificar el desmontaje seguro.
- **Sincronización:** se adoptó un esquema lectores–escritor con un mutex de "estado" y un contador de lectores protegido, evitando condiciones de carrera al leer/escribir el tablero. El master actúa como escritor exclusivo al aplicar movimientos válidos.
- **Canales de comunicación:** se usaron pipes dedicados por jugador (uno por proceso jugador) y multiplexación con select() en el master. Esto permite espera no bloqueante, manejo de timeouts y escalabilidad razonable sin hilos.
- **Planificación justa:** para la selección de turnos se implementó round-robin con una guarda para impedir que un mismo jugador juegue dos veces seguidas si existe otro jugador elegible con intento pendiente. Con esto se elimina el sesgo observado inicialmente.
- **Timeouts:** se definió un timeout/condición global de finalización (sin movimientos válidos o tablero consumido).
- **Gestión de procesos:** el master crea los procesos hijo (players y view) y maneja señales de espera (waitpid) para cierre ordenado. El unlink de la memoria compartida se realiza en el master al final, garantizando liberación única.
- **Validación de movimientos:** el master verifica límites de tablero, celdas ya consumidas, colisiones y reglas del juego antes de actualizar el estado.

Limitaciones:

- **Tolerancia a fallos acotada:** si un jugador muere inesperadamente se detecta por EOF del pipe, pero no hay "re-spawn" dinámico ni reintentos sofisticados.
- **Métricas/logs mínimos:** no persisten estadísticas ni estados intermedios para depuración post-mortem. osea si el juego falla a mitad de la ejecución, no queda un registro que te permita reconstruir qué pasó.
- **Configuración acotada:** los parámetros se leen al inicio; no hay reconfiguración en caliente (p. ej., cambiar timeouts o política de turnos).

Proceso Vista

Decisiones tomadas:

- **Visualización desacoplada:** la vista mapea game_state en modo lectura y refresca periódicamente, imprimiendo el tablero y un panel de estado (puntajes, posiciones) con códigos ANSI. Evitamos ncurses para reducir dependencias.
- **Concurrencia segura:** la lectura del estado respeta el protocolo de lectores para no interferir con las escrituras del master.
- **Usabilidad:** se agregaron colores por jugador y resúmenes compactos para diagnosticar rápidamente el avance del juego.

Limitaciones:

- **Portabilidad del terminal:** depende de secuencias ANSI; en terminales no compatibles puede haber parpadeo o artefactos.
- **Sin interacción:** la vista es pasiva; no acepta comandos ni provee historia.
- **Degradación controlada:** si la vista muere, el juego continúa, pero se pierde la observabilidad hasta reiniciarla.

Problemas Encontrados

- **Implementación de ncurses:** al momento de querer realizar una mejora en la forma en que se mostraba el estado del juego, habíamos optado por elegir ncurses. Una vez terminado, hicimos la prueba de nuevo con Valgrind y el archivo de la view estaba repleto de warnings. Eran todos por la librería ya que lo habíamos probado antes sin eso y había 0 errores. Por lo tanto, la decisión que tomamos fue hacer la view sin ncurses para evitarnos estos problemas y logramos hacer casi idéntica la vista a como estaba antes.

- **Diferencia de colores entre ChompChamps y nuestro master:** una vez que teníamos la vista nueva, con los colores y las distinciones pertinentes, decidimos probarlo con el master provisto por la cátedra. Al hacerlo, vimos que la cabeza de cada jugador y el rastro que iban dejando eran de diferentes colores. Luego de analizar lo que estaba pasando nos dimos cuenta que estábamos contando los players desde el 1, a diferencia del otro que los contaba desde el 0. Por ese motivo se produjo un desfase, y decidimos empezar a contar a los nuestros desde el número 0 para solucionar este problema.

- **Modularización**: durante el proceso de modularización encontramos dificultades al intentar unificar funciones que comparten el máster con el *player* y/o la *view*. En el caso entre *player* y *view*, la modularización resultó más sencilla, ya que las rutinas de conexión y limpieza de memoria compartida son muy similares. Sin embargo, al incluir al máster en esta lógica surgieron múltiples problemas de funcionamiento, especialmente vinculados al manejo de semáforos, memoria compartida y el *signal handler*. Esto nos obligó a retroceder parcialmente y analizar con más detalle qué partes convenía modularizar y cuáles era mejor mantener separadas, priorizando la integridad y estabilidad del sistema.

- **Errores conceptuales en el uso de funciones:** durante el desarrollo del trabajo surgieron dificultades vinculadas a la interpretación de la documentación de algunas funciones del sistema operativo. En particular, el caso más claro se dio con el uso de la función **select()**, que utilizamos para multiplexar los *pipes* de los jugadores. Al principio interpretamos de forma incorrecta los parámetros de la función, lo que nos llevó a errores de ejecución y comportamientos inesperados.

  Uno de los malentendidos más significativos estuvo en la forma de **configurar el timeout en select()**. Si bien sabíamos que tv_sec eran segundos y tv_usec microsegundos, no advertimos que ambos campos se deben **rearmar en cada iteración**, ya que la función los consume y los modifica internamente. Como consecuencia, tras la primera llamada los valores de timeout quedaban alterados y el bucle del master no respetaba el retardo esperado: en algunos casos quedaba bloqueado más tiempo de lo debido, y en otros corría casi sin pausa, generando comportamientos erráticos en el juego.

  Otro error frecuente fue con el parámetro **nfds**, donde inicialmente pasamos max_fd en lugar de max_fd + 1. Este detalle provocaba que el descriptor más alto nunca fuera monitoreado, lo que hacía que algunos jugadores quedaran sin atender en ciertos escenarios.

  Estos malentendidos evidenciaron la importancia de **leer con cuidado el manual de las funciones** y de verificar con pruebas pequeñas que los parámetros se interpreten correctamente antes de integrarlos en la lógica principal del sistema. Una vez corregidos, el juego recuperó su responsividad y se evitó la pérdida de turnos de jugadores.

Instrucciones de Compilación y Ejecución

El primer paso es clonar el repositorio de git en donde se encuentran todos los archivos necesarios para la ejecución del programa. Para hacerlo, se debe ejecutar el siguiente comando en la consola:

git clone git@github.com:FrancoBrandaa/So_TP1.git

Luego, se debe ejecutar: *docker pull agodio/itba-so-multi-platform:3.0.* Esto lo necesitamos ya que es la imagen de docker provista por la cátedra.

Una vez que se tiene todo esto, el paso siguiente es pararse dentro de la carpeta So_TP1 y allí ejecutar el comando *./execdocker.bash.* Dentro de la imagen de docker, lo que se debe hacer es moverse utilizando *cd app/src*, que es desde donde se debe ejecutar el comando make para compilar todo el proyecto.

Finalmente, con todo el proyecto compilado, se tienen 3 ejecutables con los nombres *master*, *player* y *view* respectivamente. Para ejecutar el programa y verlo por consola se debe escribir *./master* junto con una serie de parámetros que se deben elegir al momento de ejecutar. Los parámetros entre corchetes son opcionales y tienen un valor por defecto:

- [-w width]: Ancho del tablero. Default y mínimo: 10
- [-h height]: Alto del tablero. Default y mínimo: 10
- [-d delay]: milisegundos que espera el máster cada vez que se imprime el estado. Default: 200
- [-t timeout]: Timeout en segundos para recibir solicitudes de movimientos válidos. Default: 10
- [-s seed]: Semilla utilizada para la generación del tablero. Default: time(NULL)
- [-v view]: Ruta del binario de la vista. Default: Sin vista.
- -p player1 [player2]: Ruta/s de los binarios de los jugadores. Mínimo: 1, Máximo: 9.

Conclusión

Como conclusión del trabajo, se logró implementar de manera completa el juego **ChompChamps**, integrando distintos mecanismos de comunicación entre procesos vistos en la materia. Se utilizaron pipes para la interacción entre máster y jugadores, memorias compartidas para mantener el estado global del juego y semáforos para garantizar la correcta sincronización, evitando condiciones de carrera y bloqueos. Se implementaron los tres binarios independientes (máster, vista y jugador), que permiten gestionar el tablero, visualizar el estado en tiempo real y automatizar la lógica de movimiento de los jugadores.

Durante el desarrollo surgieron distintos desafíos relacionados con la concurrencia y la gestión de recursos, que fueron resueltos mediante un análisis detallado de los mecanismos POSIX y una correcta limpieza de memoria y descriptores. Surgieron además varias complicaciones durante el desarrollo del trabajo que pudimos solucionar consultando a los docentes e investigando documentación y el manual.

En definitiva, el proyecto permitió consolidar los conceptos teóricos de IPC en un entorno práctico y dinámico, cumpliendo con los requerimientos planteados.