TP1: ChompChamps

Inter Process Communication - POSIX

Grupo 3 - Integrantes:

- Ruiz, Valentín - 64046
- Allegretti, Santiago - 65531
- Migliaro, Eugenio - 65508

1. Introducción

ChompChamps es un juego multijugador tipo *snake* donde varios procesos (máster, vista y jugadores) se comunican mediante IPC POSIX. El objetivo de este trabajo práctico fue implementar el master, los jugadores y la vista, así también como su comunicación y sincronización. Para esto, se crearon procesos a partir de un proceso padre y se utilizaron pipes, semáforos y memorias compartidas para la comunicación entre los mismos. Este informe resume decisiones de diseño, cómo compilar/ejecutar el proyecto, rutas para el torneo, limitaciones y problemas encontrados con sus soluciones.

2. Arquitectura y componentes

- **Máster**: instancia y coordina los procesos, valida solicitudes de movimientos, actualiza el estado compartido, notifica a la vista y aplica política justa entre jugadores.
- **Vista**: se conecta a las memorias compartidas, imprime el estado cuando el máster lo notifica y confirma la finalización del render.
- **Jugador**: se conecta a las memorias compartidas y, tras la confirmación de que su movimiento anterior fue procesado, calcula y envía una nueva solicitud al máster mediante un pipe.

IPC y sincronización empleado

- **Memorias compartidas**: /game_state (estado del tablero y de jugadores) y /game_sync (semáforos/contadores de sincronización).
- **Semáforos anónimos**: patrón lectores-escritor para evitar inanición del escritor; semáforos punto a punto para máster↔vista y máster↔jugadores.
- **Pipes**: un pipe por jugador para enviar movimientos; lectura multiplexada en el máster.

3. Decisiones tomadas durante el desarrollo

1. **Jugador inicial (player_naive)**: en un principio se desarrolló un player_naive que simplemente analizaba sus celdas adyacentes inmediatas y se movía a la que mayor cantidad de puntos le ofrecía. Si bien no es la mejor estrategia, la idea de esto era ya tener un jugador funcional para poder probar tanto el master como la view, y luego se implementó un nuevo player más complejo.
2. **Jugador mejorado**: posteriormente se sustituyó el jugador mencionado en el punto anterior por otro con una lógica más compleja, analizando sus celdas circundantes con una profundidad mayor y haciendo un BFS para analizar ciertos parámetros, pero manteniendo el mismo contrato de IPC.
3. **Multiplexación de pipes**: se comenzó con FDs no bloqueantes pero se migró a una versión con pipes bloqueantes gestionados mediante el uso de select, para respetar la recomendación de la cátedra y simplificar la espera sobre múltiples descriptores.
4. **Parsing de argumentos**: se reemplazó strtok por getopt, siguiendo la recomendación del enunciado, mejorando robustez y claridad del manejo de flags.
5. **Sincronización máster↔vista**: se empleó un esquema de señalización bidireccional mediante los semáforos drawing_signal y not_drawing_signal, correspondientes a los semáforos A y B de la consigna, respectivamente. Esto garantiza que la vista imprime

exactamente el estado del juego que el máster acaba de modificar antes de continuar.

6. **Sincronización máster↔jugadores**: se aplicó el patrón **lectores–escritor sin inanición del escritor** para proteger el estado del juego y semáforos individuales que permiten una solicitud pendiente por jugador. Si bien esta parte fue implementada antes de la clase práctica presencial en la que se debatió este problema y su solución, la misma sirvió para verificar que la solución del problema era la correcta.

4. Problemas encontrados y cómo se resolvieron

- **Deadlock al finalizar:** surgió un problema con el orden de sincronización de los semáforos master↔view y la flag finished del estado del juego. Al finalizar el ciclo que se encargaba de toda la lógica principal del juego, nunca se le hacía un post a la view, por lo ésta nunca llegaba el ver la flag finished del game state con valor true. En consecuencia, la view nunca terminaba y al hacerle un wait_pid desde el master, éste último nunca pasaba esa línea. Para terminar el proceso, era necesario enviarle una señal para detenerlo, lo que producía leaks de memoria y era necesario borrar el docker y crear uno nuevo para poder correr nuevamente el código.

5. Instrucciones de compilación y ejecución

 # Precondición: tener la imagen local y el contenedor en ejecución.

- make docker-pull
- make docker-start

 # Entrar al contenedor

- docker exec -it os-ram bash

 # Compilación dentro del contenedor

- make host-all

 # Ejecución de ejemplo (tablero 40x12, delay=20ms, timeout=10s, 4 jugadores, semilla 123, vista)

- ./bin/master -w 40 -h 12 -d 20 -t 10 -p ./bin/player ./bin/player ./bin/player ./bin/player -s 123 -v ./bin/view

 # Limpieza de archivos

- make clean

6. Rutas relativas para el torneo

- **Vista**: ./bin/view
- **Jugador**: ./bin/player

7. Limitaciones

- **Topología del tablero**: La generación de los puntos del tablero es completamente aleatoria y no sigue ningún patrón de asignación. Para poder lograr un tablero más justo para todos los jugadores, se debería primero asignarles la posición a los mismos y luego asignarles los puntos a cada casilla del tablero, teniendo en cuenta, por ejemplo, la distancia con los jugadores.

8. Conclusiones

Se logró una implementación estable del flujo máster–vista–jugadores con sincronización correcta y comunicación eficiente.

A lo largo del desarrollo priorizamos una integración temprana entre componentes y un cierre ordenado de recursos, lo que resultó en un sistema más robusto y predecible. La experiencia nos permitió afianzar criterios de diseño en IPC, coordinación de concurrencia y señales de terminación, además de incorporar prácticas de validación que reducen bloqueos y fugas de recursos.

El proyecto también dejó aprendizajes de equipo: definir contratos claros entre procesos, documentar supuestos y revisar sistemáticamente los mecanismos de sincronización para evitar deadlocks, inanición, condiciones de carrera y espera activa.