Trabajo Práctico N1 : Inter Process Communication

Sistemas Operativos (72.11)

2025 - Primer Cuatrimestre

Integrantes:

Alexis Herrera Vegas - 64045

Sebastian Caules - 64331

Federico Kloberdanz - 62890

Introducción

Se presenta un trabajo grupal que consiste en la implementación del juego "ChompChamps" del género de "Snake", mediante el cual varios jugadores compiten por obtener la mayor cantidad de puntos repartidos en el tablero antes de quedar atrapados sin poder moverse.

Para el desarrollo del juego se aplicaron los conceptos abarcados en la primer parte de la materia acerca de los mecanismos de comunicación entre procesos (IPC) en POSIX como semáforos, pipes, memoria compartida, forks, etc.

Arquitectura del programa

El programa está dividido en 3 binarios:

master:

Aquí está la lógica del juego y la inicialización de los componentes de IPCs, siendo las memorias compartidas para el estado y la sincronización de los semáforos, los semáforos para el comienzo y finalización de la impresión del tablero y para los movimientos de los jugadores, y pipes para guardar los files descriptors para leer de los jugadores.

Por la parte de la lógica del juego, se inicializa el tablero y ubica a los jugadores, se recibe y realiza sus movimientos, se pide a la vista imprimir el nuevo estado del tablero, se determina la finalización del juego (ya sea por tiempo o porque todos los jugadores están bloqueados) y se imprime el resultado con el puntaje de cada jugador y el ganador.

player: En este binario se encuentra la inteligencia del jugador, la cual evalúa el mejor movimiento posible para luego realizarlo utilizando los semáforos y pipes para evitar colisiones entre jugadores.

view: Aquí se espera a que el master pida imprimir el tablero y se realiza eso mismo en el estado que se encuentre en ese momento. Luego se avisa que se terminó de imprimir y vuelve a la espera.

También creamos una librería con funciones que encontramos útiles para tanto el player como el master, **validateLib.c.** En esta librería hay funciones para validar si una posición es válida (está dentro del tablero y no está sobre un player), y para calcular la nueva posición de un jugador dada su posición inicial y la dirección a la cual se quiere mover. Nos pareció oportuno crear esta librería ya que nos dimos cuenta que el master debe validar y realizar los movimientos de los jugadores, y la IA del jugador debe (para ser más eficiente) verificar no moverse a una casilla inválida, y también calcular su nueva posición.

Decisiones tomadas durante el desarrollo

Si bien la estructura de los IPCs fue establecida en la consigna del trabajo, hubo varias decisiones a tomar en el desarrollo del juego.

Por ejemplo, para la vista decidimos realizar todo en la consola como recomendó la cátedra, con una vista minimalista que muestra los jugadores con sus puntajes, cantidad de movimientos y estado (bloqueado o no). Debajo está el tablero con el puntaje de cada celda y los jugadores, representados por un color. Las casillas tomadas por un jugador se llenan con su número, y su cabeza está con el fondo relleno por el color del jugador.

También, desarrollamos una inteligencia para el jugador que tiene en cuenta varios factores con pesos distintos a la hora de evaluar todos los movimientos. Entre estos factores se encuentra el puntaje de la casilla a mover, los espacios libres adyacentes que tiene para así tener mayor disponibilidad de casillas a futuro, la distancia a la pared para no encerrarse, y el riesgo de acercarse demasiado a otro jugador. Cada vez que va a moverse, el jugador evalúa todos los movimientos posibles y realiza el que mayor puntaje obtenga en la evaluación mencionada. Por la naturaleza del juego de no ser por turnos, logró ser complejo este proceso ya que hubo que buscar un balance entre encontrar el mejor movimiento y ser rápidos para obtenerlo y no perder ante una inteligencia más primitiva pero mucho más rápida.

Problemas encontrados durante el desarrollo y cómo se solucionaron.

Al momento de desarrollar el juego, observamos que algunas veces el juego terminaba por timeout, pero se quedaba congelado en lugar de imprimir lo que debería una vez terminado el juego. Luego de revisar el código y ver en qué loop o wait se quedaba colgado, descubrimos que lo que pasaba es que si varios jugadores terminaban por timeout a la vez, el primero seteaba **state->hasFinished** en true para terminar el loop del juego, pero no el loop interior a este que itera sobre los jugadores. Entonces este loop continuaba y traía problemas ya que el juego había terminado. Un simple **break** luego de setear state->hasFinished true para salir del for y que se ejecute la parte del código para terminar el programa solucionó este problema.

Una vez terminado el programa, pasamos a los testeos con Valgrind y PVS-Studio para verificar que todo esté correcto y arreglar las Warnings que daba el compilador.

Al compilar el código nos encontramos con un Warning que decía que los headColors dentro de colors.h estaban definidos pero no utilizados. Al mirar en detalle, vimos que si bien view incluye colors.h y usa ambos arrays de colores, master también lo incluye pero no usa el de headColors. Para arreglar esta única warning que nos daba sin tener que crear colors.c y complejizar el sistema de archivos, decidimos agregar el atributo __attribute__((unused)) al array headColors.

También, al testear con PVS-Studio, nos lanzaba una Warning en el fragmento:

```
unsigned int validBefore = state->players[i].requestedValidMovements;
processMovement(state->players[i].pid, move, state);
```

```
if (state->players[i].requestedValidMovements > validBefore) { . . . }
```

El warning especificaba que el if de la última línea era siempre falso, cosa que era falsa ya que sino el player no estaría haciendo movimientos válidos, o no se estarían detectando, cosa que probando el código se veía que no era el caso. Luego de indagar en cómo funciona PVS y por qué detectaría esto, vimos que no estaba detectando que al hacer hacer **processMovements** pasando por parámetro el pid del player, estabamos cambiando los requestedValidMovements del mismo player. Deducimos que esto es porque era un poco rebuscada la forma en que funcionaba processMovement, tomando de parámetro el pid del jugador, buscarlo matcheando con los pid de todos los jugadores, y actualizando sus requestedValidMovements. Por esto es que cambiamos processMovements para tomar por parámetro el player en lugar de su pid, y así simplificar el código a la vista para las personas y para PVS-Studio:

```
PlayerState *player = &state->players[i];
unsigned int validBefore = player->requestedValidMovements;
processMovement(player, move, state);
if (player->requestedValidMovements > validBefore)
```

Gracias a esto, PVS detectó que estabamos pasando el player a la función processMovement, la cual modifica sus requestedValidMovements, haciendo que si los mismos aumentaron, el if sea verdadero y se entre a su código.

Limitaciones

Una limitación al desarrollar el juego puramente en C es la vista dentro de la terminal, la cual si bien representa todo lo necesario del juego, no es la más elegante ni linda visualmente.

Por otro lado, como se explicó en las decisiones tomadas, la inteligencia de los jugadores no es lo más eficiente a la hora de encontrar el mejor movimiento por el problema de estar compitiendo contra otros jugadores que pueden ser más rápidos, así que la IA puede llegar a estar en desventaja contra otras que toman estrategias distintas.

Una limitación que descubrimos testeando el programa fue que si el master iba mucho más lento que los players, el pipe de los players se backloggeaba de movimientos (por ejemplo si quería mover hacia arriba el pipe quedaba en 0 0 0 0 0 0 0 0) entonces para cuando el master movía al jugador, el pipe tenía movimientos viejos que no correspondían al estado nuevo del tablero(0 en el ejemplo). Entonces, gracias a una respuesta de la cátedra mediante el foro a un grupo con el mismo problema, decidimos poner en el loop de player un usleep(15000) (testeamos y ese valor fue el óptimo para que no vaya tan rápido como para causar problemas ni tan lento para jugar peor al juego) para que no vaya tanto más rápido que el master y los movimientos se repitan o descoordinen. Esto es una limitación a la hora de setear el timeout como parámetro, ya que si el usuario decide ajustar el timeout y hacer el

master más lento de lo que pusimos por default, se puede volver a este problema inicial que tuvimos.

Instrucciones de compilación y ejecución

Para compilar y ejecutar el código, utilizar el comando **make**, el cual compilará los binarios de master, player y view, y también la biblioteca shmlib.o. Luego se ejecutará el comando: ./master -w -h -d <delay_ms> -t <timeout_s> -s -v ./view -p ./player [./player …]. Donde los flags representan:
-w: ancho del tablero. **-h**: alto del tablero. **-d**: delay entre turnos en milisegundos. **-t**: duración máxima del juego en segundos. **-s**: semilla para el generador aleatorio. **-v**: ruta al ejecutable de la vista. **-p**: rutas a los ejecutables de los jugadores.

Para solo compilar, usar el comando **make compile** y luego usar el comando **make run** para correrlo.

Si bien se limpian los binarios y archivos de prueba luego de cada proceso, si se termina manualmente algún proceso antes de que termine pueden llegar a quedar archivos no deseados, para esto usar el comando **make clean**, que borrará los binarios, la memoria compartida, y los archivos de testeo de PVS-Studio. También, si solo se quiere borrar la memoria compartida, usar make clean_shm, y para solo borrar los binarios usar **make clean_bin**.

Testeos con Valgrind y PVS-Studio:

Para testear memory leaks usando Valgrind, usar el comando **make valgrind-check**, el cual compila y corre el programa mediante valgrind usando los flags **--leak-check=full** y **--track-origins=yes**, imprimiendo la salida del mismo. Cabe aclarar que el testeo usará los mismos parámetros especificados en el comando **run**.

Para testear el código con PVS-Studio, usar el comando **make pvs-check**, el cual hará el “trace” al compilar, analizará el código, y hará un reporte legible en el archivo reports.tasks. Con este comando no se borra automáticamente los archivos para poder acceder al reporte, así que luego de terminar el testeo utilizar el comando **make clean_pvs** para borrar específicamente los archivos del testeo de PVS-Studio, o simplemente usar **make clean** para borrar todo.