72.11 Sistemas Operativos

Trabajo Práctico 1 - IPC

Informe

Instituto Tecnológico de Buenos Aires

Grupo 18
ChompChamps

Integrantes:

- Maria Otegui (61204) - motegui@itba.edu.ar
- German Delgado (61070) - gdelgado@itba.edu.ar
- Matias Rinaldo (60357) - mrinaldo@itba.edu.ar

Fecha de Entrega: 14-04-2025

2024

Decisiones tomadas durante el desarrollo

Durante el desarrollo de **ChompChamps**, tomamos múltiples decisiones clave que nos permitieron avanzar de forma ordenada, facilitar el debugging y garantizar el correcto funcionamiento del sistema:

- **Inicio por los procesos player.c y view.c**: Decidimos comenzar por estos dos módulos porque presentan un comportamiento más fácil de desarrollar. Esto nos permitió validar el uso de la memoria compartida y la sincronización con semáforos de forma independiente, sin depender de la lógica más compleja del proceso **master**.

- **Lógica inicial del jugador**: Para las primeras pruebas de funcionamiento utilizamos una estrategia de movimiento aleatorio. Esto nos permitió verificar rápidamente que los jugadores se conectaran correctamente y que interactuaran con el tablero según lo esperado. Luego, reemplazamos esta lógica por una más inteligente.

- **Modularización del proceso master**: A medida que la lógica del master crecía, decidimos dividir su código en **master_functions.c/h**, lo cual nos permitió:

    - Mejorar la legibilidad del código.

    - Facilitar la depuración y pruebas de funciones individuales.

    - Separar responsabilidades entre el main y la lógica interna del juego.

- **Comunicación mediante dup2() y pipes**: En lugar de implementar un protocolo de comunicación más complejo, optamos por redireccionar el stdout de los procesos player hacia el master utilizando dup2 y pipes. Esta solución fue:

    - Sencilla de implementar.

    - Eficiente y funcional para nuestro caso de uso.

    - Compatible con printf, lo que facilitó enormemente el debugging.

- **Manejo del bloqueo de jugadores**: inicialmente, cuando un jugador no encontraba movimientos válidos, simplemente llamábamos a close(1) para que el master detectara el cierre del pipe al intentar leer. Sin embargo, notamos que en algunos casos este mecanismo no era suficiente: el master no siempre detectaba el cierre de forma inmediata y algunos jugadores quedaban "activos" en pantalla pese a estar bloqueados.

Por eso, decidimos mantener una implementación más robusta y simple a la vez: Si un jugador no tiene movimientos válidos, realiza directamente un close(1) (dentro del while, y un break para salir del while) para cerrar su stdout. De esta forma, el master puede detectar que el pipe fue cerrado al leer 0 bytes, lo que indica que ese jugador está bloqueado.

- **Conversión de read_players_moves() a función booleana**: modificamos esta función para que devuelva true si algún jugador realizó un movimiento válido y false en caso contrario. Esto nos permitió implementar correctamente la política de planificación **Round Robin**, ya que el master ahora puede verificar si al menos un jugador se movió durante un ciclo y cortar el juego si todos están bloqueados.

- **Loop infinito (while(1)) en view.c**: el proceso view fue diseñado para mantenerse activo durante toda la partida, actualizando la pantalla a intervalos regulares. Utilizamos un bucle infinito controlado por semáforos para sincronizar los turnos de impresión con el master.

Instrucciones de compilación y ejecución

Compilación

Para compilar el proyecto se utiliza un makefile, lo cual permite compilar todos los binarios de forma sencilla con el siguiente comando:

```
make
```

Este comando genera los binarios master, player y view.

Ejecución

El juego se inicia ejecutando el binario master, el cual recibe diversos parámetros para configurar el entorno de juego, incluyendo el tamaño del tablero, el delay entre turnos, y las rutas a los binarios de la vista y los jugadores.

Comando de ejemplo:

```
./master -w 10 -h 10 -d 200 -t 10 -v ./view -p ./player ./player
```

Parámetros

-w: Ancho del tablero (mínimo 10)
-h: Alto del tablero (mínimo 10)
-d: Delay entre turnos (en milisegundos)
-t: Tiempo máximo de espera sin movimientos válidos (timeout, en segundos)
-v: Ruta al ejecutable de la vista
-p: Lista de rutas a los ejecutables de jugadores (mínimo 1, máximo 9)

Limitaciones y Problemas Encontrados

Durante el desarrollo del proyecto identificamos algunas limitaciones y comportamientos inesperados que afectaron la experiencia de juego y el diseño general:

Implementación del round-robin: Aunque se logró implementar una planificación round-robin entre jugadores, la solución presenta ciertas limitaciones. En algunos casos, el máster puede quedar esperando por jugadores ya bloqueados si no se detecta correctamente su inactividad, lo que genera pequeñas demoras o bloqueos temporales.

Movimiento subóptimo del jugador: A pesar de los esfuerzos por mejorar la inteligencia del jugador, este todavía realiza varios movimientos inválidos. Se priorizó la estabilidad del juego antes que optimizar el algoritmo de movimiento.

Vista desarrollada: La vista que implementamos cumple con todos los requisitos funcionales del enunciado y muestra correctamente el estado del juego. No obstante, su diseño visual difiere del binario provisto por la cátedra. A pesar de estas diferencias estéticas, fue completamente funcional para validar y testear el comportamiento del sistema.

Dependencia del orden de ejecución: En algunas ejecuciones con múltiples jugadores, el comportamiento del máster puede variar levemente dependiendo del orden en que los procesos escriben en los pipes, lo cual afecta la fluidez del round-robin.

Problemas encontrados durante el desarrollo y cómo se solucionaron

Bloqueo incorrecto de jugadores: Al inicio, los jugadores no se marcaban como bloqueados correctamente al quedarse sin movimientos. Esto impedía que el juego finalizara. Se solucionó reforzando el cierre de pipes y controlando adecuadamente la lógica de salida del bucle en el jugador.

Timeout no detectado por el máster: El máster no reconocía correctamente la inactividad prolongada. Se ajustó el uso de select() y se implementaron validaciones de tiempo para detectar la falta de movimientos válidos.

Problemas de sincronización con la vista: Al principio tuvimos problemas de sincronización entre el máster y la vista, lo que provocaba impresiones inconsistentes o fuera de orden. Esto se solucionó respetando correctamente el protocolo de semáforos propuesto en el enunciado (pending_print y print_done) junto con la lógica de lectores y escritores (readers_count_mutex, state_access_mutex, master_turn_mutex).

Dificultad para seguir la ejecución: Debido a la cantidad de procesos y sincronización, fue difícil depurar el sistema. Se resolvió utilizando herramientas como strace, valgrind y múltiples logs por proceso.

El máster no finalizaba correctamente el juego: Al bloquearse todos los jugadores, el máster no anunciaba al ganador ni finalizaba correctamente. Esto se resolvió incorporando una verificación explícita del estado de bloqueo de todos los jugadores.

Cambio de estrategia de movimiento: Se comenzó con una estrategia de movimiento aleatoria para los jugadores, que funcionaba correctamente con la versión provista del máster. Al migrar a la planificación round-robin como lo indicaba el enunciado, surgieron nuevos desafíos de sincronización y estabilidad, que requirieron ajustes en la lógica de lectura y coordinación entre procesos.

Citas de fragmentos de código reutilizados de otras fuentes

- La lógica de **lectura de argumentos con getopt** fue adaptada a partir de ejemplos disponibles en manuales de C y foros como Stack Overflow.

- La implementación de **memoria compartida y semáforos** se basó en la documentación oficial de POSIX, particularmente para el uso de shm_open, mmap, sem_init y funciones asociadas.

- El uso de **select() y manejo de pipes** para monitorear múltiples descriptores de archivo fue adaptado de ejemplos prácticos de clases de Sistemas Operativos.