72.11 Sistemas Operativos

Trabajo Práctico 1

Inter Process Communication

Instituto Tecnológico de Buenos Aires

Integrantes:

- **Keoni Lucas Duvobitsky** (62815) - kdubovitsky@itba.edu.ar
- **Nicolas Domingo Mazzitelli** (62334) - nmazzitelli@itba.edu.ar
- **Mateo Pirola Paulovich** (62810) - mpirolapaulovich@itba.edu.ar

Fecha de Entrega: 14 de septiembre de 2025

2025

Índice

1. Decisiones tomadas durante el desarrollo

En esta sección detallamos los mecanismos utilizados y las razones de su elección. Para aislar responsabilidades implementamos tres binarios: **master**, **player** y **view**. El máster concentra la coordinación (turnos, validación y actualización), el jugador produce movimientos autónomos y la vista muestra el estado en tiempo real.

Para el estado y sincronización en memoria compartida se definieron dos memorias compartidas con los nombres: SHM_STATE="/game_state" y SHM_SYNC="/game_sync". En **game_state** está el tablero, posiciones, puntajes y flags; en **game_sync** residen los semáforos y contadores necesarios para sincronizar.

Luego para **Semáforos POSIX** (A a G) y patrón lectores/escritor usamos un "handshake" entre máster y vista con A (print_needed) y B (print_done): el máster postea A cuando hay cambios y espera B hasta que la vista termine de dibujar, evitando flicker y lecturas inconsistentes.

La sincronización entre máster y jugadores se basa en **C** (mutex para evitar inanición del máster al acceder al estado), **D** (mutex para el estado del juego), **E** (mutex del contador) y **F** (cantidad de jugadores leyendo el estado). Estos elementos implementan el problema de lectores y escritores en su versión que previene la inanición del escritor. Así, múltiples jugadores pueden leer en paralelo sin bloquear indebidamente al máster cuando necesita escribir. Finalmente, cada **G[i]** es un semáforo individual que permite a cada jugador enviar un único movimiento por ciclo.

En la **comunicación máster–jugadores** implementamos pipes anónimos y multiplexación con select(). Cada jugador escribe en STDOUT_FILENO su dirección elegida (un byte de 0 a 7). El máster crea un pipe por jugador (en launch_players), redirige el stdout del hijo con dup2 y escucha simultáneamente todos los pipes con select(), lo que permite procesar lo que esté listo sin bloquearse en un jugador específico.

Para la política de turnos y equidad el bucle principal del máster aplica una política round-robin por ciclos: en cada paso (delay '-d'), atiende a lo sumo un movimiento por jugador (processed[i]), si un movimiento es invalido, se incrementa `inv_moves`; si es valido, se actualiza posicion, se suma al score y se repinta.

En el modelo del tablero y posiciones iniciales el tablero se llena con recompensas. Las posiciones iniciales se distribuyen de forma equiespaciada (distribute_positions) calculando una grilla R×C según la cantidad de jugadores, para que cada uno arranque con espacio razonable.

Sobre la estrategia del jugador (player) la IA es aleatoria: el player elige `dir = rand_r(&seed;) % 8` (las 8 direcciones en sentido horario). Antes de elegir, lee `game_over` protegido por lectores/escritor para terminar ordenadamente si el juego concluyó.

Por otro lado para la view inicializa ncurses (initscr, noecho, curs_set(0), start_color). Define pares de color con init_pair para colorear por jugador y estados, y sincroniza cada frame con el máster vía semáforos A/B. Espera A cuando hay cambios en el estado. Entra como lectora (rw_reader_enter) y copia un *snapshot* consistente. Sale rápido (rw_reader_exit) para reducir contención. Dibuja tablero y marcador, luego llama a refresh() y confirma fin del render con sem_post(B).

2. Instrucciones de compilación y ejecución

Buscamos que la ejecución sea reproducible en cualquier entorno, por eso preparamos un flujo de build y run dentro de la imagen oficial de la cátedra. Compilación (en el contenedor de la cátedra)

1. `make docker` # abre la imagen agodio/itba-so-multi-platform:3.0
2. `make deps` # instala libncurses-dev
3. `make build` # genera `./bin/master`, `./bin/player`, `./bin/view
4. `make play` # compila y ejecuta bin/play y lanza el master

Al ejecutar `make play`, hicimos un menu interactivo, en el cual podemos usar las siguientes opciones:

- enter: para editar el campo seleccionado
- r: para correr el juego con la configuración actual
- q: para salir

El juego finaliza una vez que queda un solo jugador vivo, o después de los segundos que se establezcan en nuestro menú interactivo.

Para usar el **master de la cátedra**, hay que realizar el flujo básico de ejecución y reemplazar el paso 4 por: `make run-catedra`.

3. Rutas relativas de la vista y jugador a utilizar en el torneo

El torneo requiere rutas relativas estables al binario del repositorio. En nuestra estructura de `Makefile`, los ejecutables viven en `./bin`.
• Vista: `./bin/view`
• Jugador: `./bin/player`

4. Limitaciones

Enumeramos los límites actuales que condicionan resultados:
• MAX_PLAYERS = 9 (definido en headers).
• Tableros muy chicos pueden “apretar” la distribución equiespaciada inicial.
• Si `-d` se acerca a `-t`, se ven pocas actualizaciones de la vista.
• IA aleatoria: puede quedar bloqueada y no prioriza celdas valiosas.
• La vista depende de ncurses (entorno sin soporte de terminal puede degradar la experiencia).

5. Problemas encontrados durante el desarrollo y cómo se solucionaron

Registramos incidentes, sus causas y las correcciones necesarias.
• Múltiples movimientos del mismo jugador por ciclo: se corrigió con gating `G[i]` más bandera `processed[i]` para garantizar un movimiento por ronda y jugador.
• EOF o error en pipe del jugador: al `read()==0` se marca bloqueado, se cierran FDs, evitando lecturas inconsistentes.
• Lecturas concurrentes en la view: se encapsuló acceso con helpers de lectores/escritor, priorizando al escritor para consistencia del estado.
• Falsos positivos de análisis estático: se revisaron y se validó la ejecución con valgrind sin fugas.

6. Citas de fragmentos de código reutilizados de otras fuentes

Para la realización del trabajo práctico se consultaron diversos manuales, apuntes de la cátedra y tutoriales en línea relacionados con el uso de llamadas al sistema POSIX (como fork, execve, wait, pipe, shm_open, mmap, sem_init, entre otras) y con el manejo de bibliotecas estándar en C (por ejemplo ncurses). Estos materiales se utilizaron como referencia conceptual y práctica.

En los archivos del repositorio únicamente aparecen encabezados de la herramienta de análisis estático PVS-Studio, que incluyen comentarios automáticos.

7. Conclusión

Como conclusión del trabajo, se pudieron implementar los sistemas de comunicación entre procesos estudiados en la materia, tanto pipes unidireccionales como un sistema de memoria compartida. Los problemas encontrados pudieron ser solucionados con una importante ayuda del manual, leyendo todas las macros involucradas y los valores de retorno.