Instituto Tecnológico de Buenos Aires

Trabajo Práctico Nº 1 Inter Process Communication (IPC)

Sistemas Operativos

Segundo cuatrimestre de 2025

Comisión S - Grupo 14

Integrantes:

Federico Spivak Fontaiña	60538
Jerónimo Lopez Vila	65778
Pablo Omar Germano	62837

Introducción:

El presente informe detalla el desarrollo del Trabajo Práctico Nº 1 de la materia Sistemas Operativos, centrado en los mecanismos de comunicación entre procesos (IPC) en entornos POSIX. Con el fin de adoptar y asimilar los mismos, se implementó el juego **ChompChamps** donde hasta 9 jugadores se mueven por una matriz en la que cada casilla tiene un puntaje. Cada jugador debe moverse estratégicamente para sumar la mayor cantidad de puntos y así ser el ganador. En este juego, se implementaron tres procesos distintos: el comportamiento del juego, la vista de los jugadores y el encargado de gestionar el funcionamiento de juego en general, al cual se lo denominó “master”. Estos se comunican mediante memoria compartida, semáforos y pipes, cumpliendo con los requisitos especificados en el enunciado.

Desarrollo:

Decisiones tomadas durante el desarrollo:

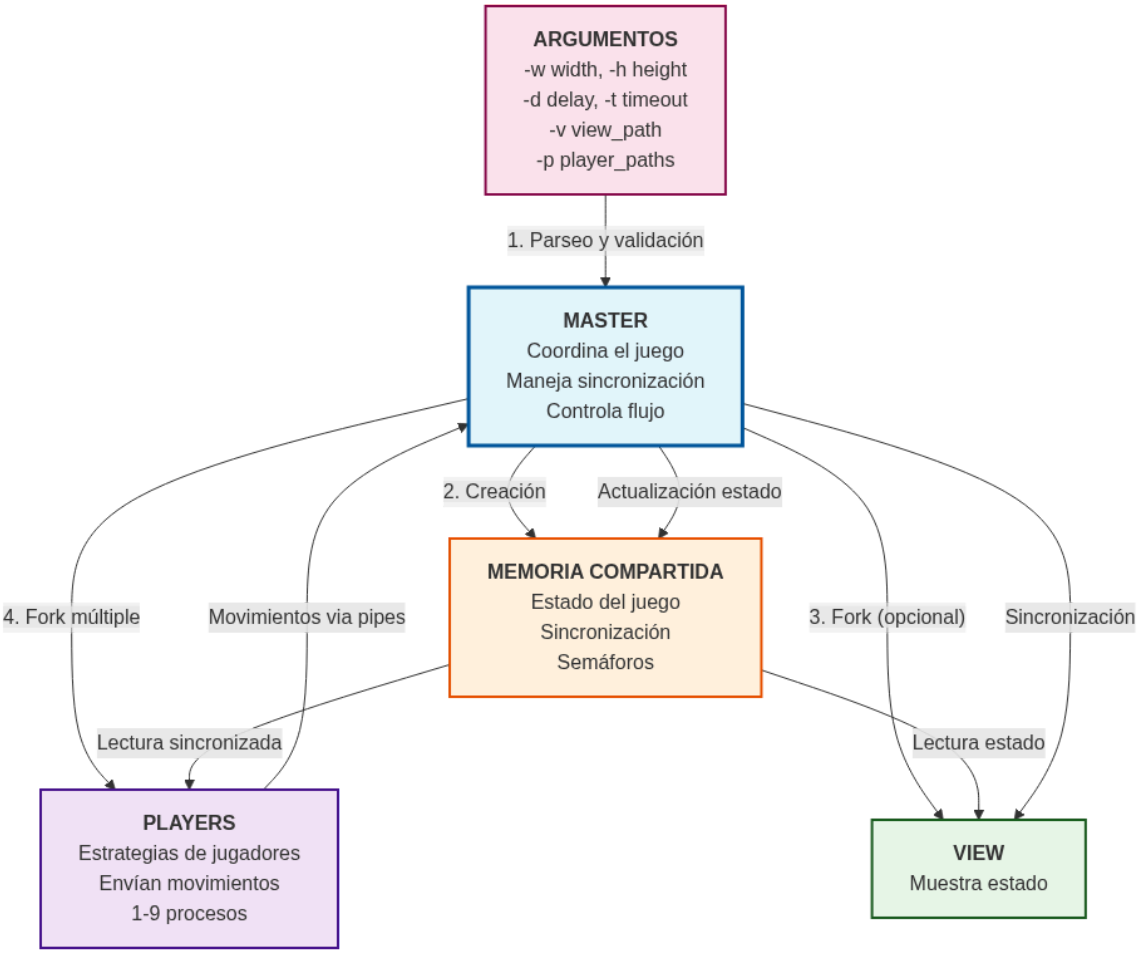
1. Validación de parámetros:

   Se validaron los parámetros del master según lo especificado en el enunciado. Los valores por defecto se aplican ante argumentos ausentes. En el caso de ingresar algún parámetro invalido, se lo notifica al usuario para que vuelva a correr el programa con los parámetros válidos. Los cuales son:

   - [-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.

2. Comunicación entre procesos:

   El máster recibe los movimientos de los jugadores mediante un pipe anónimo donde tienen el extremo de escritura vinculado al file descriptor 1. El máster crea la memoria compartida, donde se guardan los detalles de la partida, y tanto el jugador como la vista tienen permisos de lectura para ver el estado. El máster también crea la memoria compartida para semáforos, utilizados por los players para poder hacer sus movimientos, y también la vista para saber cuando debe actualizar el tablero.



Estructura de comunicación entre procesos

3. Posiciones iniciales de los jugadores:

   Para garantizar condiciones equitativas al inicio del juego, se determinó que los primeros cuatro jugadores se ubican en las esquinas del tablero y a partir del quinto jugador, se distribuyen secuencialmente en los bordes del tablero, calculando la posición de forma proporcional al tamaño del tablero y al número de jugadores restantes.

4. Se implementó un algoritmo round-robin para atender las solicitudes de movimiento de los jugadores de manera equitativa.

5. Estructuras de datos:

   Se definieron las estructuras `player_t`, `game_state_t` y `game_sync_t` para gestionar el estado del juego y la sincronización, respetando los requisitos del enunciado.

```c
// Estructura de un jugador
typedef struct {
    char name[MAX_NAME_LEN];    // Nombre del jugador
    unsigned int score;         // Puntaje
    unsigned int invalid_moves; // Cantidad de movimientos invalidos
    unsigned int valid_moves;   // Cantidad de movimientos validos
    unsigned short x, y;        // Coordenadas x e y en el tablero
    pid_t pid;                  // Identificador de proceso
    bool is_blocked;            // Indica si el jugador esta bloqueado
} player_t;
```

Estructura del jugador

```c
// Estado del juego
typedef struct {
    unsigned short width;          // Ancho del tablero
    unsigned short height;         // Alto del tablero
    unsigned int player_count;     // Cantidad de jugadores
    player_t players[MAX_PLAYERS]; // Lista de jugadores
    bool game_finished;            // Indica si el juego se ha terminado
    int board[];                   // Tablero (flexible array member)
} game_state_t;
```

Estructura del tablero

```c
// Estructura de sincronizacion
typedef struct {
    sem_t view_ready;              // Master indica a vista que hay cambios (A)
    sem_t view_done;               // Vista indica a master que termino (B)
    sem_t reader_writer_mutex;     // Mutex para evitar inanicion del master (C)
    sem_t state_mutex;             // Mutex para el estado del juego (D)
    sem_t reader_count_mutex;      // Mutex para reader_count (E)
    unsigned int reader_count;     // Cantidad de jugadores leyendo estado (F)
    sem_t player_turn[MAX_PLAYERS]; // Semaforos para cada jugador (G)
} game_sync_t;
```

Estructura de los semáforos

6. Modularización del código:

   El proyecto se estructuró de forma modular para facilitar el mantenimiento y claridad del código. Se organizó en dos carpetas principales

   - `src/`: Contiene los archivos fuente principales de cada proceso:
     - `master.c`
     - `view.c`
     - `player.c`
   - `src/lib/`: Aloja las librerías modulares compartidas que encapsulan funcionalidades específicas:
     - `common.h`: Define estructuras de datos globales y constantes compartidas.
     - `library.c/.h`: Funciones de utilidad general (manejo de errores, logging, etc.).
     - `config_management.c/.h`: Gestión de parámetros de configuración y parsing de argumentos.
     - `memory_management.c/.h`: Creación, destrucción y acceso a memoria compartida.

    - `process_management.c/.h`: Gestión de procesos (fork, cleanup, señales).
    - `game_logic.c/.h`: Lógica del juego (validación de movimientos, actualización del estado, fin del juego).
    - `view_functions.c/.h`: Funciones auxiliares para la vista (renderizado del tablero)
    - `player_functions.c/.h`: Funciones auxiliares para los jugadores (toma de decisiones, estrategias).

- `test/`: Contiene los archivos hechos por el grupo para poder testear el programa.
- Valgrind&PVS/: esta carpeta contiene los archivos utilizados para testear con Valgrind y PVS.

Además, se utilizó un `Makefile` para automatizar la compilación y enlazado de los módulos, generando los ejecutables en la carpeta `bin/`.

Instrucciones de compilación y ejecución:

Entorno de desarrollo

Para garantizar la compatibilidad y correcto funcionamiento del sistema, se utilizó el entorno de desarrollo oficial proporcionado por la cátedra mediante Docker.

Compilación con Docker

1. Inicialización del contenedor:

```
docker run -v "${PWD}:/root" --privileged -ti --name SO-TPE1
agodio/itba-so-multi-platform:3.0
```

2. **Compilación del proyecto**:

```
cd /root
make all
```

Los binarios resultantes se generan en el directorio `/root/bin/`, siendo el principal el ejecutable `master`.

Ejecución del programa

El programa principal se ejecuta con la siguiente sintaxis:

```
./bin/master [-w width] [-h height] [-d delay] [-t timeout] [-s seed]
```

```
[-v ./bin/view] -p ./bin/player1 [./bin/player2] ... [./bin/player9]
```

Ejemplos de uso:

```
# Configuración mínima
./bin/master -p ./bin/player

# Múltiples jugadores con vista
./bin/master -w 15 -h 15 -d 100 -v ./bin/view -p ./bin/player
./bin/player ./bin/player

# Configuración personalizada completa
./bin/master -w 20 -h 20 -d 50 -t 15 -s 12345 -v ./bin/view -p
./bin/player ./bin/player
```

Rutas relativas de jugador y vista para el torneo:

Para su uso en el torneo las rutas relativas de la vista y jugador son: `./bin/player` y `./bin/view`

Limitaciones:

Por cuestiones de tiempo, el jugador solo ejecuta una estrategia, a la cual se la llamó tornado. En la misma, este intenta apoderarse del centro encerrándolo en un círculo. Para esto, el player pregunta si su último movimiento es válido y así poder mantenerlo, si no lo es, intentará nuevamente con el movimiento siguiente en sentido anti-horario. La limitación llega si su única opción es el movimiento anterior al que realizó, en este caso, el player preguntará 8 veces si es un movimiento válido hasta llegar a esa opción, lo cual demora todo el proceso.Por otro lado, si no tiene movimientos válidos para ejecutar, preguntará 9 veces antes de entregar un movimiento invalido y bloquearse. Además, al plantear esa estrategia puede darse que un sector del tablero no sea consumido, lo cual son puntos que el jugador resigna.

Problemas encontrados durante el desarrollo y cómo se solucionaron:

1. **Uso de semáforos**:

   Inicialmente, hubo dificultades para entender la función de cada semáforo en la sincronización. Se resolvió mediante investigación, lectura del manual, uso del foro de la catedra y clases de consultas.

2. **Manejo de `select()` y terminación de procesos**:

   Se estudió la documentación de POSIX y se implementaron las funciones necesarias para gestionar múltiples pipes y la finalización de procesos.

3. **Utilización de Valgrind y PVS:**

   Otro de los obstáculos estuvo relacionado con el testeo, las herramientas de análisis de Valgrind y PVS, al principio llevó tiempo comprenderlas y poder llevarlas a cabo. Por ejemplo, Valgrind marcaba fugas de memoria o accesos inválidos que se entendían bien de dónde venían. Luego de leer el manual y tomar tiempo para adaptarse a estas tecnologías, se logró hacer un buen uso de las mismas.

Citas de fragmentos de código reutilizados de otras fuentes:

Para desarrollar el codigo del master y el de los jugadores, se uso como referencia el codigo mostrado en clase.

```
Master:
while(1){
    recibir_movimiento(...);

    wait(writer)
    wait(mutex)
    post(writer)

    ejecutar_movimientos(...);

    post(mutex)

    // RECORDAR CHEQUEAR EL PASAJES DE ARGUMENTOS
}

Player:
while(1){
```

```
    wait(writer)
    post(writer)

    wait(readers_count_mutex);
    if(readers_count_mutex++ == 0) wait(mutex);
    post(readers_count_mutex);



    pedir_movimiento(...)
    wait(mutex)
    enviar_movimiento(...)
    post(mutex)

}
```

Conclusión:

En conclusión, se cree haber cumplido las expectativas y requisitos del trabajo práctico, aplicando los conocimientos adquiridos en la primera parte de la materia Sistemas Operativos. Por otro lado, la consigna del mismo permitió asimilar los conceptos de una manera didáctica y práctica, permitiendo, en cierto grado, reconocer errores de implementación de los mismos.