Informe Trabajo Práctico N°1
72.11 Sistemas Operativos

Grupo 13

Lucila Borinsky	Legajo: 63039
Santiago Diaz Sieiro	Legajo: 63058
Toribio Viton Sconza	Legajo: 64275

Índice

1. Decisiones tomadas durante el desarrollo

En esta sección describimos cómo se implementó ChompChamps, haciendo hincapié en los principales mecanismos de comunicación entre procesos (IPC) y la sincronización:

- **Diseño Basado en Múltiples Procesos:** Hemos decidido separar la funcionalidad en tres programas: *máster*, *jugadores* y *vista*. De esta manera, hemos reducido la complejidad al aislar la lógica de coordinación (máster), la interacción con el tablero (jugadores) y la presentación visual (vista).

- **Uso de Memoria Compartida**
  Se utilizan dos regiones de memoria compartida que fueron creadas por ***shm_open*** y ***mmap***:

  - ➔ ***/game_state****:* Almacena la estructura GameState, la cual contiene la información global del juego (tablero, lista de jugadores, cantidad de jugadores, etc.).
  - ➔ ***/game_sync:*** Contiene la estructura GameSync con semáforos y variables auxiliares para coordinar el acceso a GameState y la impresión del tablero.

- **Semáforos para la Sincronización (POSIX)**

Dentro de ***GameSync*** se declaran varios semáforos:

- ➔ ***print_needed*** y ***print_done*** para la *sincronización entre máster y vista* (cuando el máster necesita imprimir el estado y cuando la vista ha terminado de hacerlo).
- ➔ ***turnstile*** y ***game_state_change*** para resolver el problema de lectores/escritor y evitar la inanición del máster al acceder a la memoria compartida.
- ➔ ***readers_critical_section*** y la variable ***readers*** se emplean para contar cuántos jugadores están leyendo el estado y permitir acceso seguro en modo lector/escritor.

- **Pipes entre jugadores y master**

Cada jugador se comunica con el máster mediante un ***pipe anónimo*** (creado en ***init_processes***), enviándole su movimiento. El máster usa **select()** para leer simultáneamente

múltiples pipes sin bloquearse en uno en particular. Esto permite un manejo concurrente de múltiples jugadores.

- **Distribución Inicial y Movimientos**

En ***init_board*****,** se calculan "zonas" en base a la raíz cuadrada aproximada del número de jugadores, para posicionarnos de manera algo equitativa y alejados entre sí. Cada posición inicial se marca en el tablero con valor 0 (sin recompensas) para que la primera celda no dé puntos. Esto le da dinamismo al juego sin definir una IA complicada. Puede producir bloqueos tempranos o movimientos poco "inteligentes", pero demuestran la utilización de IPC.

- **Creación de Procesos**

El máster (proceso principal) invoca: Vista (si se indica la opción ***-v***) con ***fork() + execve()*** y Jugadores (1 hasta 9) con un loop de ***fork() + execve()***, asignándoles un pipe exclusivo para el envío de movimientos.

- **Lectores/Escritor (jugadores/máster)**

Para leer y escribir la estructura ***GameState*****,** se implementó el patrón de sincronización "lectores-escritor con prioridad para el escritor". Los jugadores son "lectores" (verifican si el juego finalizó o si su estado de bloqueo cambió). El máster es "escritor" (procesa y actualiza el tablero, puntajes, bloquea jugadores). Se evitan condiciones de carrera y espera activa mediante los semáforos mencionados.

2. Instrucciones de compilación y ejecución

En primer lugar, usar la ***branch main*** del repositorio github llamado TP1_SO

- **Compilación: Makefile**

1. El archivo Makefile genera los binarios master, view y player.

2. Para compilar, simplemente correr: $make$

- **Ejecución**

En caso de querer ejecutar el programa con parámetros específicos, se detalla abajo como debe llevarse a cabo.

➔ **Máster**

El binario principal **master** se ejecuta con parámetros opcionales y obligatorios:

$./master\ [-\ w]\ [-\ h\ ]\ [-\ d\ (ms)]\ [-\ t\ (s)]\ [-\ s]\ [-\ v]\ -\ p\ [player1\_path\ ][player2\_path\ ...]$

Por ejemplo:

$./master\ -\ w\ 15\ -\ h\ 10\ -\ d\ 200\ -\ t\ 10\ -\ s\ 1234\ -\ v\ ./view\ -\ p\ ./player1\ ./player2$

➔ **Aclaración Valgrind**

Al utilizar la herramienta de análisis de memoria ***Valgrind,*** si corremos el programa con la directiva de **make run,** este arroja que quedan todavía alcanzables 143,815 bytes en 1,209 blocks en la sección de leak summary. Pero al ejecutar el programa con ***Valgrind*** pasando los paths manualmente, esto no ocurre, por lo que nuestro programa no presenta leaks de memoria.

➔ **Vista**

Si se indica -v, el máster crea un proceso vista: Se le pasan como parámetros width y heigh**t** y se conecta a **/game_state y /game_sync** para imprimir el tablero y las estadísticas según el ciclo de sincronización. No requiere invocación manual si ya se pasa el flag **-v** al máster.

➔ **Jugadores**

Cada jugador se lanza automáticamente desde master (al colocar -p). Reciben width y height, abren la memoria compartida ***/game_state y /game_sync***, y generan movimientos aleatorios (o con alguna lógica interna). Envían esos movimientos al máster por el ***STDOUT_FILENO***,

que se redirecciona al pipe.

➔ **Finalización y Limpieza**

Cuando ***game_over*** se establece en true o expira el timeout sin movimientos válidos: El máster comunica a la vista el estado final (si existe vista), luego cierra/desasocia recursos ***(shm_unlink, sem_destroy,*** etc.), y por último espera con ***waitpid*** la finalización de todos los procesos.

3. Limitaciones

1. **Número Máximo de Jugadores:** Está limitado a 9, por la definición del arreglo Player players[9].

2. **Tamaño Mínimo/Predeterminado:** El tablero exige un mínimo de 10x10 (por defecto) para que la lógica de posicionamiento no quede fuera de rango. Por debajo de ese valor, no se garantiza un correcto posicionamiento inicial.

3. **Distribución Inicial Simple:** Al dividir en "zonas" para ubicar a los jugadores, puede no ser óptima (especialmente cuando el número de jugadores es grande comparado con el tamaño del tablero).

4. **Timeout vs. Delay:** Si **delay (-d)** es mayor o muy cercano al **timeout**, el juego puede terminar antes de que se aprecien varias actualizaciones gráficas.

5. **Movimientos Básicos** Actualmente, los jugadores solo envían movimientos aleatorios. No se implementó inteligencia artificial avanzada ni razonamiento en el proceso de elección de movimiento. Esto puede generar que el jugador tome caminos poco óptimos, generando una baja cantidad de puntos y quedando bloqueado en pocos movimientos.

4. Problemas encontrados durante el desarrollo y cómo se solucionaron

1. **Todos los jugadores se movían cada turno:** se tuvo el problema que cada impresión de la vista imprimía que todos los jugadores se movían, en lugar de uno por impresión. Esto se daba porque cada turno se tomaban todos los movimientos a la vez, y se juzgaba si eran válidos al procesarlos. Para solucionarlo, en lugar de usar un solo fd_set acoplado con la macro FD_ISSET() y confirmar los movimientos enviados, se hizo un vector de fd_set (fd_set read_fds[cant_jugadores]), y cada iteración o turno se pregunta por el jugador correspondiente.

2. **Manejo de EOF/Cierre de Pipes:** Inicialmente, no se distinguía si el jugador cerraba su pipe por error o finalización. Pudimos solucionarlo, se maneja la lectura con **select() y read()**, y si cualquiera devuelve 0 o -1, marcamos al jugador como "bloqueado".

3. **Movimientos Erróneos y Posiciones Incorrectas de los Jugadores:** El ***máster*** realizaba uno o dos movimientos por turno, lo que provocaba que los jugadores no registraran correctamente sus posiciones. Para solucionarlo, cambiamos la condición de ***ready*** del player.

4. **Aclaración PVS-Studio:** Al usar la herramienta de análisis de código estático ***PVS-Studio*** este arroja la siguiente advertencia:
    - La variable game->game_over es siempre falsa: Esto no ocurre nunca al ejecutar el código, de lo contrario el programa no finalizaria nunca de correr (esta ocurre en la view y en el player).

5. Citas de fragmentos de código reutilizados

1. **Implementación de Lectores/Escritor:**

Basado en el pseudocódigo estándar de "reader-writer lock" con prioridad a escritor. Documentado en múltiples fuentes (por ejemplo, "Operating Systems: Three Easy Pieces").

2. **Uso de select()**

Guía tomada de la manpage select_tut(2) de Linux, adaptada para manejar varios file descriptores de lectura simultánea.

3. **Secuencias ANSI para Colores**

Inspirado en tutoriales de Stack Overflow y la documentación de control de consola ANSI.

4. **Posicionamiento inicial de los jugadores**

Se tomó la cuenta que determina los centros de las zonas de wikipedia.

6. Conclusión

Este proyecto de **ChompChamps** logra integrar varios mecanismos de comunicación entre procesos (memoria compartida, semáforos y pipes) y técnicas de sincronización (lectores/escritor), cumpliendo con la consigna de demostrar el uso de IPC en sistemas POSIX. Se han evitado condiciones de carrera, espera activa y/o bloqueos indeseados, y la sincronización funciona de manera estable para un número configurable de jugadores.

A pesar de las **limitaciones** detectadas, el proyecto muestra cómo estructurar y coordinar múltiples procesos concurrentes de manera ordenada. En futuras iteraciones, se podría **mejorar la IA** de los jugadores, **refinar la distribución inicial** y **optimizar** el control de tiempo y retardo, enriqueciendo la experiencia de juego.

En conclusión, **ChompChamps** sirve como un buen ejemplo práctico de cómo manejar IPCs en C bajo un entorno POSIX, aplicando sincronización y mecanismos de comunicación robustos para evitar problemas clásicos de concurrencia.