Trabajo Práctico 1

72.11 - Sistemas Operativos (SO)

1.º Cuatrimestre de 2025

Grupo 10

https://github.com/nrivas-itba/TP1-SO-ITBA-2025Q1.git

Integrantes:

Mateo Hernán Cornejo - 62722

Lautaro Bonseñor - 62842

Nicolás Rivas - 64292

Decisiones tomadas durante el desarrollo

Generales

Para el desarrollo del trabajo práctico se intentó comenzar por el jugador, después la vista y por último el máster. Se utilizó la técnica de desarrollo *pair programming*, aprovechándose de la función Live Share de Visual Studio Code, priorizando la idea de que todo el grupo entienda todos los procesos que están ocurriendo y no únicamente un aspecto en específico.

Para el uso de funciones en cualquier parte del proyecto, ya sea vista, player o master; se optó por crear una cuarta carpeta llamada "*utils*", la cual originalmente contenía únicamente *utils.h* y su implementación. Sin embargo, se decidió que era mejor incluir en la misma carpeta *ipc.h* e *ipc.c*

Ya casi finalizado el trabajo se optó por crear *ari.h*, el cual es un archivo header con únicamente los defines provistos por el binario de *ChompChamps* con texto de impresión para que muestre lo mismo.

Player

Como ya fue provisto el binario del master y la vista no es necesaria para la ejecución del mismo se decidió empezar primero con la lógica de *player*. Inicialmente, se creó con una lógica extremadamente básica para poder avanzar con lo demás, implementando movimientos aleatorios en todas las direcciones sin importar que haga un *invalid request*.

Una vez ya formalizada la vista y parte del máster se empezaron a discutir algoritmos que mejoren la toma de decisiones del jugador. Por ejemplo: el algoritmo *A** para que encuentre el camino a la celda con el puntaje más alto, una inteligencia artificial, o que calcule dos o tres pasos por delante el puntaje potencialmente más alto para decidir el siguiente movimiento. Por limitaciones temporales se tuvo que optar por el algoritmo aleatorio.

Vista

Se optó por imprimir en posiciones absolutas de la pantalla, lo que consideramos necesario. La intención de esta decisión fue generar un efecto similar a proyectos como **htop**, el cual no imprime constantemente nuevas líneas en la terminal.

A su vez, se decidió distinguir a los jugadores y sus "cabezas" por colores para lograr una interfaz más amigable con el usuario y la comprensión del estado del juego en el tablero.

En los procesos finales del desarrollo de la vista, se optó por crear *graphics.h* para el manejo de la impresión absoluta y *gameGraphics.h* para el manejo del puntaje, el tablero y los jugadores. Además de esto, también se decidió que la tabla esté ordenada descendentemente por puntaje.

Master

Como el funcionamiento equivalente al binario provisto por la cátedra era un requerimiento obligatorio se decidió decompilar el mismo con Ghidra, una herramienta de ingeniería inversa gratuita y de código abierto. Sin embargo, se nos pidió que utilicemos el binario sin símbolos. El código decompilado se puede encontrar en `/bin/originales/decompiled`

Una vez casi finalizado el código del máster, se optó por dividir el mismo para que contenga toda la lógica del juego en un archivo separado (*gameLogic*) por modularización.

Instrucciones de compilación y ejecución

Para poder compilar el proyecto, como lo indica el README.md del repositorio, es necesario:

1. Docker instalado
2. La imagen de Docker `agodio/itba-so-multi-platform:3.0`
3. (Opcional) Una licencia de PVS-Studio

Una vez descargado o clonado el repositorio, se puede ejecutar el `run.sh` preparado para que abra la imagen de docker previamente mencionada. En caso contrario, se puede inicializar la imagen normalmente mediante el comando:

```
docker run --rm -v ${PWD}:/root --security-opt seccomp:unconfined -it
                agodio/itba-so-multi-platform:3.0
```

Una vez dentro de la terminal, hay que entrar al directorio `ROOT` mediante `cd ~` y ejecutar el comando `make all`, el cual compilará todos los archivos necesarios dentro del directorio `/bin`. Para poder ejecutarlo con mayor facilidad se proporcionó el comando `make run` con los siguientes parámetros:

- w: Ancho del tablero. Default y mínimo: 10
- h: Alto del tablero. Default y mínimo: 10
- d: milisegundos que espera el máster cada vez que se imprime el estado. Default: 200
- t: Timeout en segundos para recibir solicitudes de movimientos válidos. Default: 10
- p: Cantidad de jugadores. Default y mínimo: 1
- m: Qué tipo de master quiere utilizar (si el provisto por la cátedra ("original") o el que fue hecho por el grupo). Default: "custom" (creado por el grupo)

Ejemplo de uso:

```
make run w=11 h=12 p=3
make run w=11 m=original
```

Este comando automáticamente compila y corre el juego, utilizando la vista y los jugadores hechos por el grupo y la seed por defecto aclarada por la cátedra. Su único propósito es facilitar la ejecución del programa, en caso de no utilizarlo, también se puede optar por el comando usual con los mismos parámetros provistos por la cátedra.

```
./bin/master.o -p ./bin/player10.o -v ./bin/view.o
```

- [-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.

Uso de Valgrind y PVS-Studio

Para poder utilizar la herramienta de Valgrind en el proyecto, al igual que con la ejecución, se puede utilizar por comodidad el comando provisto `make run_valgrind`, con los mismos parámetros que su contraparte `make run`. En su defecto, una vez compilado el proyecto, también se puede optar por el comando

```
valgrind ./bin/master.o -p ./bin/player10.o -v ./bin/view.o
```

Para poder utilizar el analizador de PVS-Studio, primero es necesario validar las credenciales del programa, mediante el comando

```
pvs-studio-analyzer credentials "<Your User>" "XXXX-XXXX-XXXX-XXXX"
```

Una vez validado, simplemente ejecute el comando `make pvs` para que genere los archivos deseados.

Limitaciones

Para la impresión absoluta se había considerado usar la librería *Ncurses*, la cual provee una API que permite escribir interfaces basadas en texto (TUIs). Sin embargo, esta daba un error *"unknown"* en la línea de inicialización. Por lo que se optó por implementar una solución que utiliza códigos de escape ANSI.

Como el sizeof se procesa en tiempo de compilación y las macros en el tiempo de procesado, la macro *stringize* no puede "stringificar" sizeof. Por lo tanto, esto provoca que no se pueda usar sizeof para calcular el tamaño de un string, específicamente en el puntaje de un jugador y la cantidad de solicitudes inválidas.

También, ante cualquier error de ejecución todos los procesos creados abortan instantáneamente sin desconectarse de las shared memories ni realizar ningún tipo de tarea de limpieza de recursos. Se contempló la posibilidad de solucionar esto y, aunque si bien en clase se habló de la posibilidad de usar atexit, se nos recomendó no solucionar este problema.

Por otro lado, el master original imprime en forma incorrecta el valor de retorno de los *players* y *view*. Si bien esto se puede solucionar utilizando la macro `WEXITSTATUS`, se decidió no solucionarlo en nuestra implementación ya que, de acuerdo a lo hablado por mail, su intención es ver únicamente si el valor de retorno es 0 o no.

Asimismo, si bien se mencionó que *player* fue el primer aspecto que se hizo a la hora de empezar con el desarrollo del proyecto este fue el último en pulirse. Consideramos que la vista y el máster eran más vitales para el trabajo, es por esto que se trabajó a contrarreloj para poder mejorar el algoritmo del jugador.

De menor importancia, el *signal handler* de `SIGWINCH`, que se encarga de manejar el cambio de tamaño de la terminal, no vuelve a imprimir el tablero. Por lo tanto, se nota un pequeño parpadeo cada vez que se modifica el tamaño de la pantalla. A su vez, también se nota que hay algunas terminales que no soportan emojis, por lo que hay partes de la vista que lucen como un cuadrado vacío en las mismas.

Por último, como se utilizó una licencia gratuita de PVS-Studio, este se utilizó limitado a sus funcionalidades básicas, en contraste al potencial que tiene su licencia paga.

Problemas encontrados y su solución

Generales

El principal problema que tuvimos fue la modularización del código, ya que originalmente existía entre 4 archivos: *master, utils, player10* y *view*. Esto dificulta comprender el proceso que hace cada función, lo cual se solucionó como se menciona en “Decisiones tomadas durante el desarrollo”: creando *ipc.h, gameLogic.h, graphics.h, gameGraphics.h, ari.h* y sus implementaciones.

Master

Se dificultó el entendimiento del código decompilado original sin símbolos. Se solucionó renombrando cada aparición de una función o variable por lo que se asumía que hacía en base a los conocimientos aprendidos en clase. Todo esto se hizo de una manera muy informal en una copia de ese decompilado en `/bin/originales/decompiled/cc-copy.c`

Otro problema menor fue el procesamiento de argumentos y la creación de la *shared memory*, la que no nos permitía crearla directamente con los detalles del juego. Estos últimos se tenían que copiar en ella después de que sea creada, lo cual se considera un problema de eficiencia.

Asimismo, también se notó un problema con el orden en el que se procesaban los argumentos, específicamente con el *delay* y la *view*. Si no ingresaban estos argumentos el juego indicaba que se ignorará *delay* pues no hay vista, lo cual se notó que no ocurre en el binario original (no debería, pues no hay delay para ignorar). Por lo tanto, se consideró hacer dos *getopt()* separados, el cual se consideró que podría solucionar el problema de la shared memory, pero al final se optó por un simple booleano que detecta si el delay fue declarado para decidir si se muestra o no el mensaje.

Vista

El principal problema encontrado en su desarrollo fue la implementación de Ncurses y encontrar una alternativa al mismo ante la incapacidad de hacer que funcione. Esto fue solucionado a partir de mover la posición del cursor a la posición deseada dentro de una *screen_t*. El struct *screen_t* indica el alto y ancho de un contenedor, y su offset vertical y horizontal en relación a la posición de origen de la terminal. Toda esta solución se puede ver en la carpeta *graphics*.

Otro problema encontrado fue el posible cambio de tamaño de la terminal y que la misma vista sea *“responsive”*. Esto se soluciona usando la librería *fcntl.h*, el cual puede indicar el tamaño de la consola y, en base a ello, calculamos cuántos caracteres ocupa una celda para poder imprimir.

A partir de la intención de que se adapte al cambio de tamaño de la terminal, se generó un problema en el que al realizar dicha acción la vista presentaba errores gráficos, como la repetición de parte del tablero o de sus bordes. Esto se soluciona con limpiar la pantalla cada vez que se actualiza el tamaño de la terminal con la señal `SIGWINCH`.

Player

El mayor problema relacionado al desarrollo de *player* fue la solución de inteligencia artificial. Se llevó a cabo esta solución ya que era la que sonaba más interesante y con más potencial, sin embargo, se notó que no daba los resultados esperados y que requiere muchos más ciclos de

entrenamiento; además que el entrenamiento era muy costoso para el CPU. Es por esto que se terminó optando por la decisión aleatoria que, irónicamente, daba resultados similares.

Citas de código de fuente externa

- La función `numLen` fue copiada de https://github.com/nrivas-itba/PI-TPE/blob/main/utils.c.
- Gran parte del código del *máster* son modificaciones del código decompilado de la consigna.
- La elección de colores de los jugadores fue preguntada a ChatGPT