UNIVERSITÀ di VERONA | Dipartimento di INFORMATICA

Laboratorio di Sistemi Operativi

Elaborato System Call

A.A 2022/2023

Studenti:

Vittorio Maria Stano (VR457793)

Stefano Esposito (VR461049)

14/06/2023

INDICE GENERALE

DESCRIZIONE GENERALE DEL PROGETTO

Il progetto "Forza 4" è un'applicazione scritta in linguaggio C che implementa il gioco da tavolo "Forza 4" utilizzando le system call SYSTEMV. L'applicazione funziona su sistemi operativi UNIX/LINUX e coinvolge due giocatori che si sfidano su un campo di gioco quadrato o rettangolare.
Lo scopo del gioco è allineare quattro gettoni dello stesso giocatore in verticale, orizzontale o diagonale per vincere.

Il gioco è composto da due eseguibili: F4Server e F4Client.

F4Server si occupa di inizializzare il gioco e arbitrare la partita tra i due giocatori. Viene eseguito con il seguente comando:
./F4Server <num_righe> <num_colonne> <gettone_player1> <gettone_player2>

Dove i parametri specificano le dimensioni del campo di gioco e i gettoni utilizzati dai due giocatori.
Il server gestisce anche eventuali errori di esecuzione legati alla presenza di campi di gioco precedenti.

In caso di doppia pressione del tasto CTRL-C, il server termina il gioco correttamente avvisando i processi dei giocatori che il gioco è stato interrotto esternamente.

Il server "arbitra" la partita, controllando se un giocatore ha vinto dopo ogni mossa e segnalando il vincitore ai client. Il server notifica anche quando non sono possibili ulteriori inserimenti di gettoni, indicando la fine del gioco con un pareggio. Quando uno dei due giocatori vince, la partita termina per entrambi i giocatori.

F4Client rappresenta il singolo giocatore e si occupa di raccogliere la mossa del giocatore e visualizzare il campo di gioco. Viene eseguito con il seguente comando:
./F4Client <nome_utente>

Il server rimane in attesa di entrambi i giocatori prima di iniziare il gioco. Una volta avviata la partita, il client mostra il campo di gioco aggiornato e richiede al giocatore la colonna nella quale inserire il gettone.
Il client gestisce anche la pressione del tasto CTRL-C, segnalando la sconfitta del giocatore per abbandono.

Time-out per ogni mossa
All'avvio del server, viene definito un numero di secondi di time-out entro cui ogni client deve effettuare la propria mossa. Se un giocatore non effettua una mossa entro il tempo stabilito, il turno passa all'altro giocatore senza che il primo abbia giocato. Anche in questo caso il player viene notificato.

Giocatore automatico

Un client può essere avviato come giocatore automatico, generando casualmente la colonna in cui inserire il gettone. Se la colonna scelta è già piena, viene generata una nuova colonna finché non ne viene trovata una valida.

Viene eseguito con il seguente comando:

./F4Client <nome_utente> *

Quando viene invocato in questa modalità, il client genera un processo figlio che avvia un processo client standard utilizzando il nome utente specificato dalla riga di comando **<nome_utente>**, oltre al processo client automatico richiesto.

F4-SERVER

"F4Server.c" è il file sorgente principale per l'implementazione gioco "Forza 4". Di seguito una breve descrizione delle attività svolte dal codice:

1. **Controllo del formato della riga di comando:** viene eseguito un controllo sul formato dei parametri passati alla riga di comando.
2. **Gestione dei segnali:** i segnali *SIGINT*, *SIGALRM* e SIGABRT vengono impostati per essere gestiti, per una successiva implementazione.
3. **Fase di sincronizzazione:** vengono sincronizzati i due client attraverso l'utilizzo di una *FIFO*.
4. **Creazione del campo di gioco:** utilizzando la memoria condivisa, viene creato il campo da gioco, il quale viene inizializzato con spazi vuoti per il successivo riempimento con i token dei giocatori, da parte dei client.
5. **Sincronizzazione del gioco:** il server gestisce la partita, interagendo con i due client. Vengono inviati segnali ai client per consentire loro di giocare entro un limite di tempo ("*TIMEOUT*"). In seguito, viene verificata la condizione di vincita o di pareggio.
6. **Pulizia delle risorse:** vengono inviati segnali di terminazione ai client, rimosse le risorse IPC (memoria condivisa, semafori) ed eliminati i file temporanei utilizzati durante il gioco.

Il file "F4Server.c" implementa quindi la logica del server per il gioco e gestisce la comunicazione con i client attraverso l'uso di IPC e segnali.

Il server è stato implementato con la gestione della doppia pressione del CTRL-C per consentire una chiusura controllata della partita. Quando viene premuto il CTRL-C la prima volta, il server avvia un countdown e stampa un messaggio che avvisa l'utente che un'ulteriore pressione comporta la conclusione della partita.

Se viene premuto nuovamente il CTRL-C entro il tempo specificato, il server esegue le seguenti azioni:

1. **Termina la partita**: esegue le azioni necessarie per terminare correttamente la partita in corso.

2. **Chiude i client**: comunica ai due client che la partita è stata interrotta e richiede la loro chiusura, indicando che la chiusura della partita è stata causata esternamente.
3. **Rimozione dei file e delle IPC create**: rimuove tutti i file e le *interprocess communication* (IPC) create durante l'avvio del programma per ripristinare le risorse del sistema.
4. **Termina il server**: il server viene chiuso completamente.

Di seguito sono indicati i segnali gestiti dal server con le relative descrizioni:

- **SIGABRT**: utilizzato per gestire la doppia pressione del *CTRL-C* lato client. Quando viene ricevuto il segnale, viene chiamata la funzione *clientClosure*, la quale termina i processi client, chiude il server, rimuove le IPC e i file creati e avvisa l'utente che ha premuto il doppio *CTRL-C* che ha perso per abbandono.
- **SIGINT**: utilizzato per gestire l'interruzione di sistema, generata dalla pressione del tasto *CTRL-C,* lato client. Quando il segnale viene rilevato, viene eseguita la funzione "*sendWarning*", che avverte che un'ulteriore pressione del tasto *CTRL-C* comporta il termine del programma.
- **SIGALRM**: utilizzato per gestire il tempo entro il quale può avvenire l'inserimento della colonna da parte del giocatore attuale. Se non avviene entro il tempo stabilito, viene chiamata la funzione *trackSignal*, la quale imposta la variabile "*receivedSignal*" a *SIGALRM*. Successivamente si passa il turno all'altro giocatore.

F4-CLIENT

"F4Client.c" è un'implementazione dei giocatori nel gioco "Forza 4". Di seguito una breve descrizione delle attività svolte dal codice:

1. **Controllo del formato della riga di comando**: il client riceve il nome del giocatore come argomento della riga di comando e verifica la correttezza di tali parametri.
2. **Gestione dei segnali**: vengono impostati i gestori di segnale per diverse condizioni, come *SIGINT, SIGUSR1, SIGUSR2, SIGHUP, SIGABRT* e *SIGTERM*.
3. **Fase di sincronizzazione**: il client scrive il suo PID nella FIFO condivisa con il server e legge a sua volta il PID del server dal FIFO. Inoltre, dalla FIFO, riceve anche i parametri di gioco quali: row e column che rappresentano le dimensioni del campo da gioco e il token del giocatore. Inoltre, identifica anche l'ID della memoria condivisa e del set di semafori utilizzati.
4. **Configurazione dell'ambiente di gioco**: imposta l'ambiente di gioco, incluso il collegamento alla memoria condivisa.
5. **Ciclo di gioco**: il client entra in un ciclo infinito per il proprio turno di gioco:
    A. Mostra il campo di gioco.
    B. Avvia un conto alla rovescia in una finestra separata (F4Countdown), per dare visualizzazione al giocatore del tempo rimanente per eseguire una mossa.
    C. Permette al giocatore di inserire il proprio token nel campo di gioco.
    D. Gestisce eventuali segnali di time-out o fine del gioco.

E. Mostra il campo di gioco aggiornato.
F. Attende il turno dell'avversario.

Il processo continua a ripetersi fino alla fine del gioco o alla ricezione di un segnale di uscita.

Di seguito sono indicati i segnali gestiti dal client con le relative descrizioni:

- **SIGINT**: utilizzato per gestire l'interruzione di sistema, generata dalla pressione del tasto *CTRL-C,* lato client. Quando il segnale viene rilevato, viene eseguita la funzione "*sendWarning*", che avverte che un'ulteriore pressione del tasto *CTRL-C* comporta il termine del programma.
- **SIGUSR1**: utilizzato per segnalare al client che ha vinto la partita corrente. Quando il segnale viene rilevato, viene eseguita la funzione "*clientWin*", la quale stampa a video la vincita del rispettivo client.
- **SIGUSR2**: utilizzato per gestire il turno del giocatore. Viene chiamata la funzione *trackSignal*, la quale imposta la variabile "*receivedSignal*" a *SIGUSR2*. Successivamente si sblocca il turno per il giocatore corrente.
- **SIGHUP**: utilizzato per gestire l'abbandono della partita da parte dell'altro giocatore. Quando il segnale viene rilevato, viene eseguita la funzione "*opponentAbandoned*", la quale stampa a video l'abbandono dell'altro client.
- **SIGABRT**: utilizzato per gestire la chiusura del server dopo una doppia pressione del tasto *CTRL-C* da parte del client. Quando il segnale viene rilevato, viene eseguita la funzione "*serverClosure*", la quale avvisa il server che a sua volta chiude entrambi i client, sé stesso e conclude la partita.
- **SIGTERM**: utilizzato per segnalare al client che ha perso la partita corrente. Quando il segnale viene rilevato, viene eseguita la funzione "*clientLose*", la quale stampa a video la sconfitta del rispettivo client.

F4-COUNTDOWN

"F4Countdown.c" implementa un countdown per consentire al giocatore di inserire una mossa entro un determinato limite di tempo.
Utilizza il segnale *SIGALRM* per la gestione stessa del conto alla rovescia. Viene visualizzato il tempo rimanente nel terminale con diversi colori a seconda del tempo rimanente. Se il giocatore non inserisce una mossa entro il tempo limite, il programma termina.
Il codice avvia un timer che viene decrementato ad intervalli regolari. Ogni volta che il timer viene aggiornato, l'output viene visualizzato nella finestra xterm, mostrando il tempo rimanente per effettuare la mossa.
L'implementazione del countdown è basata su un ciclo *while* utilizzando la funzione *pause()*.

In sintesi, questo programma gestisce il countdown e fornisce un riscontro visivo al giocatore sul tempo rimanente per effettuare una mossa.

F4-AUTOPLAYER

"F4AutoPlayer.c" implementa il giocatore automatico all'interno del "Forza 4". Il principio di funzionamento è simile a "F4Client.c", dettagliato in precedenza, ma con l'implementazione della logica per il player automatico.
Il programma gestisce i segnali per la comunicazione con il server e l'altro giocatore "umano". Viene eseguita una fase di sincronizzazione tramite una FIFO condivisa e viene inizializzato l'ambiente di gioco.
Nel ciclo di gioco principale, il giocatore automatico attende il proprio turno, durante il quale inserisce il proprio token nel campo di gioco tramite l'utilizzo di funzioni specifiche, analizzate nelle pagine che seguono.
Successivamente, viene mostrato nuovamente il campo di gioco e il programma attende la mossa dell'avversario. Questo ciclo continua fino al termine della partita.

FUNZIONI UTILIZZATE

Nel progetto sono state implementate le seguenti funzioni per garantire il corretto funzionamento del gioco e gestire eventuali errori di esecuzione:

FUNZIONI SERVER (server_function):

- **void serverCommandLineCheck(int, char *[]):** questa funzione controlla la stringa di avvio del programma *F4Server*. Prende come argomenti un intero che rappresenta il numero di parametri passati e un array di stringhe che contiene i parametri stessi. La funzione verifica la correttezza dei parametri, come da specifica, fornendo un messaggio di errore in caso di parametri errati. Successivamente si avvisa mostrando un esempio di sequenza corretta per agevolare l'avvio.
- **void sendWarning(int):** viene utilizzata per avvisare l'utente dopo la prima pressione del tasto *CTRL+C*. Prende come argomento un intero che rappresenta il segnale ricevuto. La sua implementazione consente di inviare un avviso appropriato al server per l'eventuale interruzione del gioco.
- **void continueGame(int):** utilizzata per continuare il gioco dopo che è scaduto il tempo a seguito della prima pressione del tasto *CTRL+C*. La funzione prende come argomento un intero che rappresenta il segnale ricevuto. La sua implementazione consente di gestire in modo appropriato la continuazione del gioco dopo un'interruzione.
- **void endGame(int):** viene chiamata per terminare il gioco dopo una doppia pressione del tasto *CTRL+C,* entro un tempo prestabilito, nel caso di questo progetto fissato a 3 secondi. Prende come argomento un intero che rappresenta il segnale ricevuto. La sua implementazione consente di gestire la terminazione corretta del gioco e notificare agli altri giocatori la vittoria per abbandono.
- **winIndex winCheck(char, char(*)[]):** questa funzione viene utilizzata per controllare se un giocatore ha vinto la partita. Prende come argomenti un carattere che rappresenta il simbolo

del giocatore e un puntatore a un array bidimensionale che rappresenta il campo di gioco. La funzione controlla se ci sono quattro gettoni allineati orizzontalmente, verticalmente o diagonalmente, restituendo un oggetto "*winIndex*" che contiene le coordinate del gettone vincente, per una successiva implementazione in *showWinPlayingField*.

- **void showWinPlayingField(int, int, char (*)[], winIndex):** visualizza il campo di gioco della partita in cui è avvenuta una vittoria. Prende in input un array bidimensionale che rappresenta il campo di gioco e una struttura *winIndex* chiamata "*cell*" contenente le informazioni sulla posizione della vittoria. Questo per rappresentare a terminale la posizione della vincita nel campo da gioco in dettaglio con il colore verde. La stampa avviene lato server.
- **void trackSignal(int):** utilizzata per assegnare la variabile di controllo per il server al segnale *SIGALRM* per terminare il ciclo e il relativo countdown, nel turno del client corrente.
- **void removeFile(char *):** utilizzata per rimuovere un file specificato dal suo nome contribuendo alla gestione e all'eliminazione dei file all'interno di un programma o di un'applicazione tramite la chiamata *unlink*, la quale rimuove il collegamento. Dopo la rimozione, lo spazio occupato dal file nel sistema sarà recuperato.
- **void clientClosure(void):** viene utilizzata per gestire la conclusione della partita nel client dopo l'abbandono del giocatore. Prende come argomento un intero che rappresenta il segnale ricevuto. Quando viene ricevuto il segnale *SIGUSR1*, viene stampato il messaggio di sconfitta per abbandono e il programma viene terminato.

FUNZIONI CLIENT (client_function):

- **int clientCommandLineCheck(int, char *[]):** come in *server_function*, ma implementata lato client.
- **void sendWarning(int):** come in *server_function*, ma implementata lato client.
- **void continueGame(int):** come in *server_function*, ma implementata lato client.
- **int insertToken(char (*)[]):** viene utilizzata per inserire un gettone nel campo di gioco. Prende come argomenti un puntatore a un array bidimensionale che rappresenta il campo da gioco. La funzione verifica se la mossa è valida e se è possibile inserire il gettone in quella posizione. Per richiedere l'inserimento della colonna, viene creato ogni volta un processo figlio che si occupa dell'inserimento. Restituisce 0 se la colonna è piena e non è possibile inserire il gettone, se la colonna richiesta non esiste oppure se il valore inserito non è un intero. Restituisce 1 se l'inserimento è andato a buon fine e -1 se non è avvenuto alcun inserimento.
- **void showPlayingField(char (*)[]):** viene chiamata per stampare il campo di gioco aggiornato. Prende come parametro un puntatore a un array bidimensionale che rappresenta il campo da gioco. La funzione visualizza il campo di gioco aggiornato volta per volta con tutte le mosse effettuate fino a quel momento.
- **void opponentAbbandoned(int):** utilizzata per avvisare l'abbandono da parte dell'avversario dalla partita corrente. Prende come argomento un intero che rappresenta il segnale ricevuto e stampa a video la rispettiva stringa di vincita e chiusura del client.

- **void youAbbandoned(int):** utilizzata per avvisare la sconfitta per abbandono da parte del client corrente. Prende come argomento un intero che rappresenta il segnale ricevuto e stampa a video la rispettiva stringa di sconfitta e chiusura del client.
- **void serverClosure(int):** prende come argomento un intero che rappresenta il segnale ricevuto e viene chiamata quando il server invia un segnale di chiusura.
- **void trackSignal(int):** utilizzata per assegnare la variabile di controllo per il server al segnale *SIGUSR2* per attendere il proprio turno.
- **void clienWin(int):** utilizzata per notificare la vincita al client corrente. Prende come argomento un intero che rappresenta il segnale ricevuto.
- **void clientLose(int):** utilizzata per notificare la sconfitta al client corrente. Prende come argomento un intero che rappresenta il segnale ricevuto.

Nell'header del file *client_function*, si ha la dichiarazione di una struttura di tipo ***pipeInfo*** che viene passata attraverso una *PIPE*, essa contiene i seguenti campi:

- **_Bool readCheck:** flag per verificare la lettura del valore colonna inserito dal giocatore.
- **char cPlayerColumn[100]:** array per memorizzare l'informazione della Colonna inserita dal giocatore.

ALTRE FUNZIONI CONDIVISE (function):

- **void setSignalSet(_Bool []):** viene chiamata per configurare i segnali attraverso il parametro *"_Bool signals[]",* in modo che solo determinati segnali vengano bloccati, mentre gli altri rimangono attivi per il processo corrente. Ogni volta che viene invocata, imposta i segnali specificati nel parametro.
- **void itoc(int, char *):** converte un numero intero (param. "*intero*") in una stringa di caratteri (param **childPid*). Calcola la lunghezza del numero intero e lo converte in caratteri memorizzando l'informazione nel puntatore fornito.
  Viene utilizzata all'interno della funzione *writePid* per convertire l'ID del processo in una stringa di caratteri, la quale viene poi scritta tramite write() nel file specificato
- **void readPid(char *):** legge l'ID del processo da un file e lo restituisce come un intero. È utile per recuperare l'ID del processo da un file quando è necessario utilizzarlo in altre zone del programma.
- **void writePid(char *, int):** converte l'ID del processo in una stringa e lo scrive in un file specificato. È utile per memorizzare l'ID del processo in un file in modo che possa essere utilizzato successivamente da altre parti del programma.
- **int searchFile(char *):** utilizzata per la ricerca di un file specificato nella directory corrente, restituendo un valore che indica se il file è stato trovato (0) o meno (-1). Viene utilizzata per gestire eventuali condizioni errate per l'avvio del client o server (es. apertura di due server oppure apertura di un client umano e successivamente di un client automatico)
- **void nothing(int):** la funzione non ha alcuna funzione ma si occupa di mandare a buon fine la ricezione del segnale *SIGALARM* in *client_function*. Viene utilizzata da F4Countdown.c.

Nell'header del file *function*, si ha la dichiarazione di una struttura di tipo ***info*** che viene passata attraverso una *FIFO*, essa contiene i seguenti campi:

- **int pid:** rappresenta il PID del processo F4Server.
- **int shmid:** rappresenta l'ID della *shared memory*, dove è inizializzato il campo da gioco.
- **int semid:** rappresenta l'ID del semaforo da utilizzare.
- **char playerToken:** contiene il gettone del giocatore del client connesso in quel momento.
- **int row:** rappresenta il numero di righe del campo da gioco.
- **int column:** rappresenta il numero di colonne del campo da gioco.

La struttura di tipo ***winIndex*** invece è utilizzata per memorizzare le coordinate di vincita del player. Viene utilizzata da *showWinPlayingField* per la stampa della sequenza vincente di colore verde. La struttura contiene le seguenti dichiarazioni:

- **int row[4]:** utilizzato per la memorizzazione delle coordinate di riga.
- **Int column[4]:** utilizzato per la memorizzazione delle coordinate di colonna.
- **Int returnValue:** valore di controllo utilizzato per la stampa di colore verde.

Nel file sono dichiarate alcune variabili di tipo **extern**. Questo metodo è utile per dichiarare una variabile che è stata definita in un altro file sorgente del programma.
Queste variabili consentono la condivisione e l'utilizzo di queste tra più file. È un meccanismo per evitare la duplicazione delle variabili e per permettere la comunicazione tra diverse parti del programma.

In questo file header sono definiti anche i valori dei semafori utilizzati, il tempo per l'inserimento del token e il tempo per la pressione del doppio CTRL-C.

FUNZIONI FIFO (fifo):

- **createFifo(char *):** questa funzione è utilizzata per la creazione di una *FIFO* (*First In First Out*) bloccante. Riceve come parametro il percorso della *FIFO* da creare ("*pathFifo*"). All'utente proprietario viene autorizzata la lettura e la scrittura, mentre per l'intero gruppo solo la scrittura. Se la creazione fallisce, viene stampato un messaggio di errore.

FUNZIONI SHARED MEMORY (shared_memory):

- **int createSharedMemory(size_t):** utilizzata per la creazione di una memoria condivisa utilizzata per memorizzare la matrice del campo di gioco. Prende in input la dimensione desiderata per la memoria condivisa e restituisce l'identificatore della memoria condivisa creata.
- **void *attachSharedMemory(int, int):** utilizzata per associare la memoria condivisa identificata da "*shared_memory_id*" all'indirizzo di memoria del processo chiamante. Inoltre, prende in input l'identificatore della memoria condivisa e un intero che rappresenta il numero di colonne del campo da gioco. Restituisce un puntatore all'area di memoria condivisa.
- **void detachSharedMemory(int, char(*)[]):** utilizzata per dissociare la memoria condivisa identificata da "*shared_memory_id*" dall'indirizzo di memoria del processo chiamante. Prende

in input il numero di colonne del campo di gioco e il puntatore all' array bidimensionale che rappresenta il campo stesso.

- **void removeSharedMemory(int):** questa funzione è responsabile della rimozione della memoria condivisa identificata da "*shared_memory_id*". Viene utilizzata per de allocare la memoria condivisa utilizzata per la memorizzazione della matrice del campo di gioco. Prende in input l'identificatore della memoria condivisa.

FUNZIONI SEMAFORI (semaphore):

- **int createSemaphoreSet(void):** utilizzata per creare un set di semafori che gestisce la sincronizzazione e la protezione delle zone critiche del progetto definendo i valori iniziali. Restituisce l'identificatore del set di semafori creato.
- **void setSemaphore(int, unsigned short, short):** utilizzata per eseguire operazioni sui semafori all'interno di un insieme specificato dall'identificatore "*semid*", utilizzando l'indice del semaforo "*numSem*" e l'operazione da eseguire "*semOp*".
- **void removeSemaphoreSet(int):** utilizzata per rimuovere un set di semafori indicato da *semid* utilizzando la funzione *semctl*.

Nell'header del file *semaphore*, si ha la dichiarazione di ***union semun***, utilizzata per la gestione dei parametri per le operazioni sui semafori IPC (Inter-Process Communication) utilizzando la funzione *semctl*.

La struttura *union semun* contiene tre membri:

- **int value**: utilizzato per impostare o ottenere il valore di un singolo semaforo all'interno di un insieme di semafori.
- **struct semid_ds *buffer**: puntatore a una struttura *semid_ds* che viene utilizzata per ottenere o impostare informazioni sullo stato dell'insieme di semafori.
- **unsigned short *semaphoreSet**: puntatore a un array di semafori, utilizzato per ottenere o impostare il valore di tutti i semafori all'interno dell'insieme.

FUNZIONI ERR_EXIT:

- **void errExit(const char *message):** stampa un messaggio di errore, rimuove le risorse *IPC* create e i file specificati, terminando il programma con uno stato di errore. È utilizzata per gestire un errore causato "bruscamente" e pulire tutto l'ambiente prima di terminare l'esecuzione. Prende in input una stringa che rappresenta l'errore da stampare.
- **void errnoCheck(int):** viene utilizzata per stampare a terminale un dettaglio relativo al valore di "*errno*" passato come parametro. Viene utilizzata per gestire gli errori di tipo "*errno*" e fornisce informazioni aggiuntive utili per la diagnostica degli errori durante l'esecuzione. Prende in input il codice errore da stampare.

FUNZIONI CLIENT AUTOMATICO (autoPlayer_function):

- **void sendWarning(int)**
- **void continueGame(int)**
- **int insertToken(char (*)[])**
- **void showPlayingField(char (*)[])**
- **void opponentAbbandoned(int)**
- **void youAbbandoned(int)**
- **void serverClosure(int)**
- **void trackSignal(int)**
- **void clienWin(int)**
- **void clientLose(int)**

Queste funzioni sono implementate allo stesso modo di quelle indicate in *client_function*.
Sono presenti ulteriori funzioni per implementare l'algoritmo del giocatore automatico in modo da eseguire diverse valutazioni al fine di prendere decisioni strategiche per portare alla vincita della partita in modo "intelligente".
Viene creata una matrice per l'algoritmo per effettuare tutte le valutazioni. La matrice viene definita board e come dimensioni prende le dimensioni della matrice definita all'avvio di F4Server.

- **int is_valid_move(char (*)[], int):** verifica se una mossa in una determinata colonna del campo da gioco è valida. Restituisce 1 se la colonna è vuota, altrimenti 0.
- **int make_move(char (*)[], int, char):** effettua una mossa. Inserisce il simbolo del giocatore nella colonna specificata e restituisce il numero di riga in cui è stata effettuata la mossa. Se la mossa non è valida, restituisce -1.
- **int evaluate_window(char *):** valuta una finestra di 4 caselle all'interno del campo da gioco. Conta il numero di simboli del giocatore corrente, del giocatore avversario e delle caselle vuote nella finestra. Restituisce un punteggio in base alla situazione della finestra, indicando vincite, mosse vincenti potenziali o doppie mosse potenziali.
- **int score_position(char (*)[]):** calcola il punteggio complessivo di una posizione del campo da gioco. Valuta le righe, le colonne e le diagonali del campo da gioco, utilizzando la funzione.
- **int find_best_move(char (*)[], int):** trova la migliore mossa da effettuare per il giocatore automatico. La funzione itera su tutte le colonne del campo da gioco, verifica se una mossa è valida, simula la mossa utilizzando la funzione *make_move* (definita in precedenza), valuta la posizione risultante utilizzando l'algoritmo *minimax*, e annulla la mossa effettuata. Restituisce la colonna corrispondente alla migliore mossa trovata.
- **int minimax(char (*)[], int, int, int, int):** implementa l'algoritmo minimax per valutare le posizioni del campo da gioco e trovare la mossa ottimale. Valuta le posizioni in modo ricorsivo, alternando il giocatore corrente e l'avversario. Utilizza i valori *alpha* e *beta* per il pruning dell'albero di ricerca, migliorando l'efficienza dell'algoritmo.
- **int max(int, int):** restituisce il massimo tra i due rispettivi valori di input (funzione di supporto).
- **int min(int, int):** restituisce il minimo tra i due rispettivi valori di input (funzione di supporto).

TECNICA PRUNING

Il **pruning**, nell'ambito degli algoritmi di ricerca, è una tecnica utilizzata per ridurre il numero di rami di un albero di ricerca che devono essere esplorati. Consiste nell'eliminare o ignorare determinati rami dell'albero che sono considerati non promettenti o non rilevanti per la soluzione del problema.

Può essere applicato in diversi contesti, ma uno dei suoi utilizzi più comuni è nell'algoritmo **minimax** con **pruning alpha-beta**. Questo algoritmo viene utilizzato per valutare le posizioni di un gioco a turni con informazione perfetta, come gli scacchi o il Forza 4. L'obiettivo dell'algoritmo è trovare la mossa migliore da fare in una determinata posizione del gioco.

Durante l'esplorazione dell'albero di ricerca, l'algoritmo **minimax** valuta le mosse possibili e assegna loro un valore di utilità. Durante questo processo, vengono mantenuti due valori, chiamati "**alpha**" e "**beta**", che rappresentano rispettivamente il valore migliore attualmente conosciuto per il giocatore massimizzante (**alpha**) e il valore migliore attualmente conosciuto per il giocatore minimizzante (**beta**).

Quando l'algoritmo **minimax** incontra una mossa che determina chiaramente un valore peggiore rispetto a una mossa precedente, può escludere ulteriori esplorazioni dei rami successivi a quella mossa, poiché il giocatore avversario non sceglierà mai quella mossa. Questo è possibile perché il giocatore avversario cercherà sempre di minimizzare il valore dell'utilità.

Attraverso il **pruning alpha-beta**, l'algoritmo **minimax** può saltare intere porzioni dell'albero di ricerca, riducendo così significativamente il numero di mosse da valutare. Ciò comporta un notevole risparmio di tempo computazionale, rendendo l'algoritmo più efficiente.

NOTE CONCLUSIVE

È importante notare che è perfettamente normale che l'algoritmo del giocatore automatico impieghi un po' di tempo per generare una mossa promettente. L'algoritmo di ricerca utilizzato può richiedere una considerevole quantità di tempo di calcolo, specialmente quando la complessità del problema aumenta.

La velocità di calcolo dell'algoritmo dipenderà da diversi fattori, tra cui la potenza di calcolo del sistema su cui viene eseguito, la dimensione del problema e il numero di mosse possibili da valutare. In alcuni casi, potrebbe essere necessario aumentare la complessità dell'algoritmo per ottenere mosse di qualità migliore, aggiornando il parametro **MAX_DEPTH** nel file *autoPlayer_function.h*. Ciò comporterà anche un aumento del tempo di calcolo richiesto per generare una mossa.

È importante bilanciare la complessità con il tempo di calcolo disponibile per garantire una buona esperienza di gioco.