UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E COMPUTAÇÃO

Projeto Final de Sistemas Operacionais:

Jogo Connect Four implementado em C/C++, utilizando Threads, Semáforos e alocação manual de Memória.

Leonardo Giovanni Prati - 8300079
Luca Machado Bottino - 9760300

SÃO CARLOS
NOVEMBRO DE 2018

Sumário

1 Introdução

1.1 Objetivo

O presente trabalho tem por objetivo implementar e entender os mecanismos de um sistema operacional de alocação de memória, threads e semáforos. Para isso, foi desenvolvido um software do jogo Connect Four em C/C++ que engloba esses 3 conteúdos em sua mecânica e permite jogar tanto em modo single como mult player.

1.2 O jogo - Connect Four (Liga Quatro)

O jogo Connect Four é uma espécie de jogo da velha, que consiste em fichas vermelhas e azuis e um tabuleiro bilateral e vertical de 7 linhas por 6 colunas. O objetivo do jogo é ir colocando as fichas, até que o jogador consiga colocar 4 fichas em linha (podendo ser na horizontal, na vertical ou na diagonal), e impedir que o adversário consiga o mesmo. Quem conseguir completar uma linha de quatro itens primeiro é o vencedor.

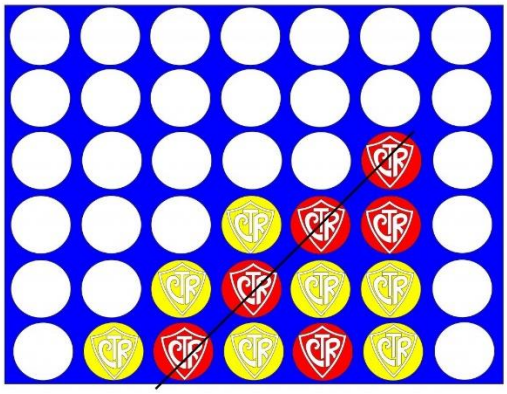

Figura 1 – Exemplo do objetivo do jogo

Fonte: Teaching Stripling Warriors

2 Implementação

2.1 Versão 2 Players

A versão de 2 jogadores do jogo foi implementada de modo que pudesse ser jogada em um terminal Linux. Utilizando um tabuleiro de 7 x 6 cada jogador realiza um movimento por vez. O jogador escolhe a coluna que quer inserir a peça e essa peça “cai” até a parte mais baixa do tabuleiro. Uma função chamada check_win realiza a verificação do final do jogo. Se o jogo não tiver terminado é passada a vez para o outro jogador.

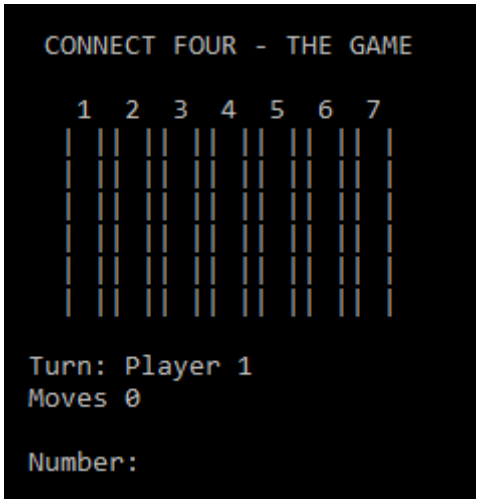

Figura 2 – Imagem da Implementação do Jogo Connect Four.

2.2 Memoria – Visão Geral

Para entender o funcionamento da alocação de memória foram desenvolvidas duas funções básicas utilizadas pelo programa: a função malloc_modified e a função free_modified. Assim como as funções malloc e free comumente utilizadas na programação C/C++ essas funções alocam e liberam uma parte da memória tendo, em alto nível, a mesma funcionalidade das funções clássicas.

2.3 Implementação da memória de baixo nível – função srbk.

Para desenvolver as funções de alocação de memória utilizou-se da função sbrk(). A função sbrk retorna um ponteiro void para o espaço de memória em que se encontra o final da memória heap a crescido do argumento da função em bytes. Ou seja, sbrk(0) retorna um ponteiro para o final da heap e sbrk(1024) retorna um ponteiro para um endereço de memória 1KB acima da heap.

2.4 Malloc

A função void *malloc_modified (size_t size) recebe como argumento quantos bytes o programa gostaria de alocar e retorna um ponteiro void para o inicio da memoria alocada. Para isso, utiliza-se uma variável mutex que bloqueia o programa quando for feita a locação de memória. A alocação é uma segmentação de memória de tamanho size em bytes não necessariamente continua de memória, conforme mostra a Figura 3 abaixo.

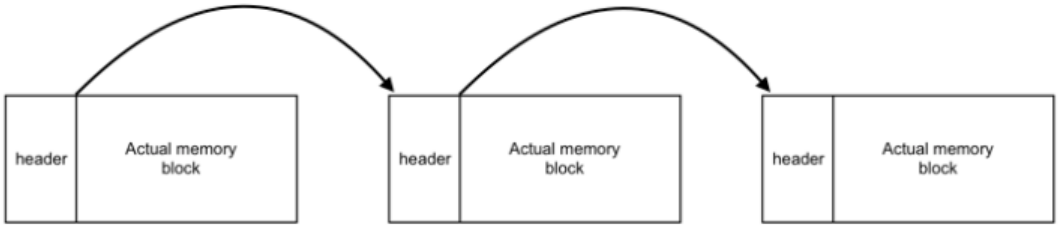

Figura 3 – Exemplo de alocação de memória.

A função node *search_empty_memory(size_t size) procura pela próxima posição de memória livre que comporta o tamanho de meória que está tentando ser alocado. Se o espaço alocado foi encontrado, a função malloc marca o bloco de memória como is_empty = 1, alocando o espaço de memória e retornando um ponteiro para o ínicio da memória alocada. Deste modo, podemos dizer que esse malloc utiliza do algoritimo First Fit, uma vez que o primeiro espaço de

memória que comporta o tamanho size que se procura alocar é reservado para o programa que solicitou a função malloc.

2.5 Free

Análogamente a função malloc a função free_modified desaloca os blocos de memória previamente alocados na heap. Essa função primeiro tranca a variável global do mutex e então procura pelo ponteiro final da heap utilizando a função sbrk. Então a função procura pelo local em que foi alocado a memória que se deseja liberar e marca a variável is_empty como 1, liberando o espaço para ser posteriormente alocado por outra variável. Se o espaço não foi encontrado a função retorna NULL. Após a desalocação o mutex é liberado.

2.6 Avaliação de Jogada

O modo um jogador é baseado na execução de uma função de avaliação de jogadas, que escolherá qual a melhor posição para o Computador. A função de avaliação de jogadas é o número de jogadas que faltam para o Jogador ativo vencer o jogo. Um número positivo 2 mostra que o jogador ativo vencerá na sua segunda jogada a partir da avaliação. Se o resultado da avaliação for zero, o jogo será um empate. Se a avaliação é negativa, o jogador perderá naquele número de rodadas.

A avaliação de jogadas foi retirada de um repositório[9] de terceiros no github. Optou-se por não implementar o algoritmo de avaliação, conforme descrito na proposta do projeto, uma vez que a implementação deste solver foge do escopo do trabalho e eleva a complexidade do código.

Esta condição de vitória é calculada por uma busca na árvore de jogadas disponíveis. Cada nó é uma posição no tabuleiro. Um nó possui um filho para cada coluna disponível para se jogar a partir dele.

A árvore é percorrida através do algoritmo Minimax. Este algoritmo percorre as jogadas recursivamente, retornando um valor positivo quando encontra uma jogada vencedora, igual a pontuação daquela jogada (o número de movimentos restantes para o fim do jogo). Caso encontre um movimento em que o oponente vence, retorna um valor negativo igual a pontuação daquela jogada.

Como o número de jogadas disponíveis é muito grande, utiliza-se a técnica alpha-beta pruning para reduzir o espaço de busca. Estabelece-se uma janela de pontuação. Qualquer jogada avaliada com valor menor que a janela é descartada, enquanto qualquer pontuação maior é utilizada para redefinir os limites da janela. Assim, evita-se aprofundar a exploração nos ramos da árvore que levam à posições ruins.

2.7 Semáforos e Threads

Como estratégia de execução da busca, criou-se dois buffers compartilhados, implementados como FIFO. Cada buffer contém mutexes e semáforos associados, protegendo os buffers contra acesso concorrente e registrando a contagem de itens.

Para preencher e consumir os itens dos buffers, implementou-se duas classes, Producer e Consumer. Enquanto Producer recebe as jogadas disponíveis para o PC e as coloca no buffer de jogadas, Consumer continuamente tenta retirar jogadas do buffer para processar, e coloca as pontuações calculadas no buffer de resultados.

Vale notar que o programa percorre a árvore até o fim (ignorando os caminhos excluidos pelo pruning), portanto o algoritmo resolve o jogo perfeitamente. Connect four é um jogo matematicamente resolvido, e sabe-se que é possível ganhar sempre sendo o primeiro jogador. Além disso, sempre é possível empatar sendo o segundo jogador.

3 Conclusão

O jogo Connect Four é um jogo que, apesar de lúdico, aborda diversos conceitos pertinentes à Matemática e à Ciência da Computação. Deste modo, nesta versão do jogo Connect Four, foi desenvolvido uma função que aloca manualmente um segmento de memória e outra que desaloca esse mesmo segmento dado. Uma IA avalia jogadas no modo single player através de uma busca pela árvore de jogadas possíveis, e tenta encontrar o melhor movimento. Além disso, threads foram disparadas para avaliar qual dos movimentos da IA produziriam o melhor resultado, processando paralelamente, a árvore de jogadas. Este processamento é sincronizado através de threads e mutexes, evitando conflitos ao acessar recursos compartilhados na memória.

No desenvolvimento deste software foi possível aplicar os conceitos de programação em C/C++, alocação de memória, threads e sincronização, contribuindo para o entendimento da complexidade dos problemas que o SO enfrenta para gerenciar os recursos para os usuários, das soluções que surgiram ao longo da história dos Sistemas Operacionais e, de forma geral, propiciando um entendimento prático da disciplina.

4 Referências

1. Linux Programmer Manual – PTHREADS (7) Disponível em: <http://man7.org/linux/man-pages/man7/pthreads.7.html> Acesso em: 28/11/2018.
2. James Golick – Memory Allocators 101 Disponível em: <https://www.jamesgolick.com/2013/5/15/memory-allocators-101> Acesso em: 28/11/2018.
3. Code Project – Pritam Zope – Writting Our Own Simple Memory Manager in C/C++. Disponível em: <https://www.codeproject.com/Articles/1180826/Writing-Our-Own-Simple-Memory-Manager-In-C-Cpluspl>. Acesso em: 28/11/2018.
4. Linux Programmer Manual – MMAP (2) Disponível em: <http://man7.org/linux/man-pages/man2/mmap.2.html> Acesso em: 28/11/2018.
5. Arhen Sreedharan – Memory Allocators 101 – Write a simple memory allocator. Disponível em: <https://arjunsreedharan.org/post/148675821737/memory-allocators-101-write-a-simple-memory> Acesso em: 28/11/2018.
6. Solving Connect 4: How to build a perfect AI. Disponível em: <http://blog.gamesolver.org/>. Acesso em 28/11/2018.
7. C++ Multithreading. Disponível em: <https://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm>. Acesso em: 28/11/2018
8. C++ Reference. Disponível em: <http://www.cplusplus.com/reference/>. Acesso em: 28/11/2018
9. Pascal Pons, Connect 4 Solver – Disponível em: <https://github.com/PascalPons/connect4>. Acesso em: 28/11/2018.