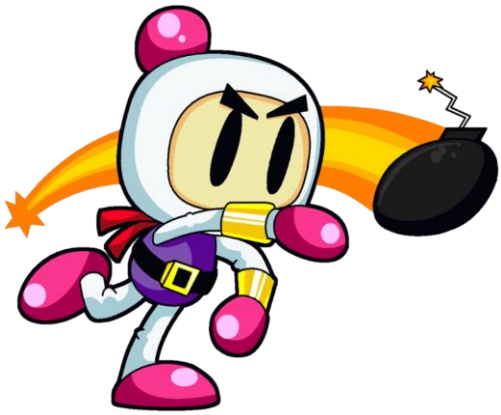
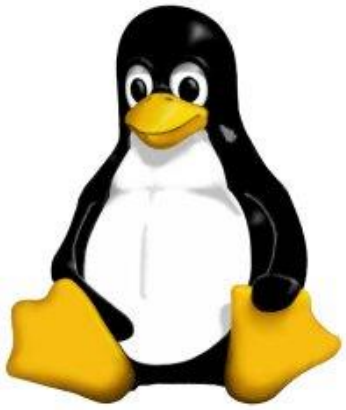
Licenciatura de Engenharia Informática
Instituto Superior de Engenharia de Coimbra
Ano letivo 2017/2018





Bomberman

Trabalho prático Meta 3

Sistemas Operativos – 2017/2018

Relatório Final realizado por:
Marco Duarte nº21260412 P2
Marco Lopes nº21260401 P2

Índice

Introdução

No âmbito da unidade curricular de Sistemas Operativos foi proposto a realização de um jogo (Bomberman) para Linux, de forma aprender na prática consolidar todos os conceitos de programação para UNIX. O contexto do jogo é multijogador na mesma máquina o que implica a comunicação entre processos neste caso entre cliente/servidor através de named pipes para que seja possível criar um programa dinâmico com múltiplos jogadores conectados a um único servidor.

Objetivos

Mantém-se da meta 1 os seguintes objetivos:

- Criação das estruturas de dados;
- Guardar o username e password dos clientes num ficheiro de texto;
- Leitura e validação dos comandos do servidor.

Os novos objetivos para a segunda meta são:

- Comunicação com Named Pipes;
- Gestão dos Clientes;
- Login dos jogadores.

Objetivos finais:

- Todas as funcionalidades

Estruturas de Dados

Foram criadas inicialmente 6 estruturas necessárias para a criação do jogo e o seu bom funcionamento. Estas são relativas aos jogadores, inimigos, objetos, partidas e pedidos.

Código

```c
#define L 21
#define C 61
#define nInim 5
#define nObj 0

#define ESC 27

#define SERVER_FIFO "/tmp/s_fifo"
#define CLIENT_FIFO "/tmp/cliente_%d_fifo"


WINDOW *win_menu;
WINDOW *win_game;

typedef struct labirinto Labirinto;
typedef struct jogador Jogador;
typedef struct inimigo Inimigo;
typedef struct objeto Objeto;
typedef struct partida Partida;
typedef struct posicao Posicao;
typedef struct pedido Pedido;

struct posicao{
    int x, y;
};

struct objeto{
    Posicao pos;
    char tipo;
    int contObj;
};

struct pedido{
    char comando[20];
};

struct jogador{
    Posicao pos;
    char caracter;
    char username[20];
    char password[20];
    int PID;
    int pontos;
    int vidas;
    int login;
    Objeto obj[2];
    Pedido p;
}jogadores;
```

```c
struct inimigo{
    Posicao pos;
    int contInim;
    char id;
};


struct partida{
    Jogador j[20];
    char mat[L][C]; // LABIRINTO!
    int tempoJogo;
    int n_ativos;
    int fim;
};
```

Informação e Justificação

Como primeira estrutura foi criada a Posicao que vai ser usada em outras estruturas e visa ajudar na criação de coordenadas(x,y) para certos elementos do jogo.

Seguidamente criamos estruturas para os objetos do labirinto com a posição da mesma, uma variável para contar os objetos e também o tipo. Optamos pela criação da mais importante que é a estrutura Jogador. Nesta adicionamos uma variável da estrutura Posicao para as coordenadas dos jogadores no labirinto, duas strings para guardar o username e a password, duas variáveis inteiras para os pontos e para as vidas, variável inteira para verificar o estado do jogador na partida, e uma variável caracter para definir a "cara" do jogador. Além disso contém uma estrutura com 2 elementos para os dois tipos de bombas e um pedido para transmitir informações entre cliente e servidor (movimento).

A próxima estrutura também pertencente ao conteúdo do labirinto é a Inimigo com uma variável Posicao para obtenção de coordenadas e uma variável inteira para contagem dos inimigos presentes no labirinto e um id.

Por fim, mas não menos importante, foi criada uma estrutura para o labirinto, com um array dos jogadores, uma matriz para desenhar o labirinto, o tempo de jogo, e numero de jogadores ativos no jogo.

Sinais

São usados dos tipos de sinais: “SIGUSR1” e “SIGUSR2” para comunicar entre os clientes e servidores. O objetivo dos sinais é de informar o cliente que saiu do jogo por algum motivo (expulso ou o servidor terminou) e então terminar de forma ordeira. Os sinais são enviados sempre pelo Servidor aos clientes ativos. O servidor envia um dos sinais sempre que quer tirar um cliente do jogo (voluntariamente ou não), por exemplo quando o servidor termina, ou quando algum jogador morre.
Quando o sinal é recebido pelo cliente, este interpreta-o e fecha os fifos e termina-os antes de desligar o programa.

Considerações Gerais

Caso o administrador tente encerrar o Servidor brutamente, o mesmo avisa que vai ser encerrado e desliga de forma correta, fechando todos os pipes.
Ao desligar o Servidor de forma correta através do comando *shutdown,* este envia uma mensagem a todos os Clientes através de um sinal, e cada Cliente interpreta esse sinal e desliga de forma correta, informando o utilizador do mesmo.

De forma a possibilitar que o Servidor consiga lidar com a informação que vem dos pipes de todos os clientes e ao “mesmo tempo” interpretar os comandos e realizar as suas tarefas foi usado o mecanismo **select.** Pela mesma razão foi usado também um **select** no Cliente.

Importante:
Como o labirinto é gerado de forma aleatória sempre que o servidor é iniciado, por vezes pode acontecer de algum jogador ou inimigo calhar de ficarem presos por paredes indestrutíveis. O cliente pode muito bem sair e voltar a entrar, e esperar que sejam atribuídas melhores coordenadas desta vez. Quando ao inimigo, se um jogador o conseguir libertar, o mesmo move-se normalmente.

Arquitetura e Estratégia de Named Pipes

De forma a estabelecer uma comunicação entre processos: o Servidor e os Clientes foram usados named pipes (fifo). O Servidor é composto (cria) por um fifo geral que é conhecido por todos os clientes e estes abrem e escrevem enquanto que o Servidor fica em permanente leitura.
Enquanto que para haver uma comunicação no sentido Servidor-Cliente, o servidor abre o fifo de cada cliente para enviar a informação necessária. O nome desse fifo é obtido pelo PID de cada Cliente que foi enviado na comunicação Cliente-Servidor. De uma forma geral, o que diferencia os fifos dos Clientes é o PID.

Nos named pipes são passadas estruturas do tipo Partida e do tipo Jogador.

Threads usadas

No lado do cliente não é usada nenhuma thread, uma vez que usamos e contornamos a necessidade de tratamento de várias fontes por parte do cliente com o mecanismo select.

Já no servidor são usadas 3 tipos de threads. Uma para a explosão de uma bomba pequena, outra para a explosão de uma bomba grande e ainda outra para cada inimigo do jogo. Entendemos que para estas situações o uso mais indicado são mesmo as threads de forma a permitir uma tarefa em simultâneo a todas as outras do programa.

O objetivo das threads para as explosões, como dá a entender, é a realização de uma explosão ao mesmo tempo que possa acontecer o resto do jogo (movimentações, envio do mapa para todos os clientes, outras explosões, verificações, etc.) em simultâneo. O seu funcionamento é muito simples: realiza um sleep de 2 segundos, faz um ciclo de acordo com a dimensão da bomba, e preenche em sua volta um caracter especial de forma a possibilitar ver que posições afetou, de seguida volta a fazer um sleep de 2 segundos e preenche as mesmas posições com espaços vazios. Como é óbvio, as paredes indestrutíveis continuam iguais. É criada quando um cliente decide colocar uma bomba, e é terminada assim que acaba de fazer a explosão.

As threads para os inimigos servem para dar uma certa vida aos inimigos, como o próprio nome indica. Com isto, a thread é responsável pelo movimento de cada inimigo (fazendo as mesmas verificações que é preciso fazer para os jogadores) e também pela sua morte. Quando são atingidos por uma bomba de um jogador, pode acontecer o seguinte:

- Largar uma bomba pequena (30%)
- Largar uma bomba grande (20%)
- Largar uma vida (40%)
- Efeito surpresa – Transforma-se numa bomba suicida gigante (10%)

Neste caso as threads são criadas no inicio do programa logo após o servidor gerar o labirinto, e são terminadas quando cada inimigo morre ou então quando o servidor termina.

Mecanismos de sincronização

O mecanismo de sincronização usado são os mutex. Apenas usamos para as threads dos inimigos, porque entendemos que se usássemos para a das bombas, apenas quando uma bomba acabasse de explodir é que era possível colocar outra para explodir.

Sendo assim, na movimentação dos inimigos usamos mutex de forma a garantir que um inimigo só se move quando o anterior tiver acabado de fazer a sua movimentação/funções. Este mecanismo permite manter uma consistência nos dados dos inimigos e assim não haja problemas com a parte lógica do jogo.

Gestão de Clientes / Login

Para realizar uma melhor gestão de todos os clientes do jogo foi implementada memória dinâmica no lado do Servidor. Sempre que um cliente passa de forma positiva pelo processo de login é imediatamente guardada toda a sua informação num array dinâmico do tipo Jogador. De igual forma quando o Cliente sai do programa (voluntariamente ou não) é liberta a memória relativa ao mesmo.
Um Cliente pode sair do programa voluntariamente ou então pode ser expulso por qualquer motivo pelo administrador. Neste caso o mesmo é retirado do array, e é enviada uma mensagem ao Cliente a informando-o do acontecido. Esta mensagem é enviada através de sinais. O Cliente tem a capacidade de interpretar esses sinais e verificar que foi expulso da sessão.

Há um processo pelo qual todos os clientes têm que passar antes de entrar no jogo, desde a validação dos mesmos (username e password) no ficheiro de texto gerido pelo Servidor e também a verificação da existência do jogador no jogo.

Na versão final do jogo, ao mesmo tempo que é guardada a informação dos jogadores logados no array dinâmico, é feita também a cópia para uma estrutura do tipo Partida de forma a enviar a partida com os jogadores todos aos clientes.

Funcionalidades Realizadas

O estado de todas as funcionalidades pedidas até ao momento é o seguinte:

Funcionalidade	Não Implementada	Parcialmente Implementada	Totalmente Implementada
Estrutura de Dados			X
Armazenamento dos pares Username/Password			X
Leitura e Validação dos comandos do servidor			X
Comunicação Named Pipes			X
Gestão dos Clientes			X
Login dos jogadores			X
Lógica do Jogo			X
Visão atualizada do jogo			X
Inimigos			X
Explosões			X
Carregar o mapa de um txt		X	
Níveis	X		

Verificação e validação

De forma a garantir um bom funcionamento do programa em muitas situações são realizadas verificações para ver se corre tudo bem e então o programa pode prosseguir com as suas tarefas, como por exemplo:

- A falta de um arquivo txt com os jogadores ao correr o programa
- Verificar se já existe um servidor em execução (tanto do lado do servidor como do cliente)
- Validação se são configurados de forma correta o tratamento dos sinais
- Verificação de todos os named pipes
- Validação de todos os comados do admin
- Verificação após as escritas nos pipes de forma a garantir que foi enviada toda a informação do servidor para o cliente e vice-versa
- Etc...

Comportamentos anómalos conhecidos

Quando um novo cliente entra no jogo, é preciso carregar numa tecla de movimento de forma a conseguir ver o jogador, no entanto se não mexer ele atualiza o labirinto normalmente. O mesmo acontece quando é por exemplo atingido por uma bomba (precisa de mover de novo).