Sistemas Operacionais I

quem quer ser um milionário?

Cesar Guibo

Leonardo Fonseca

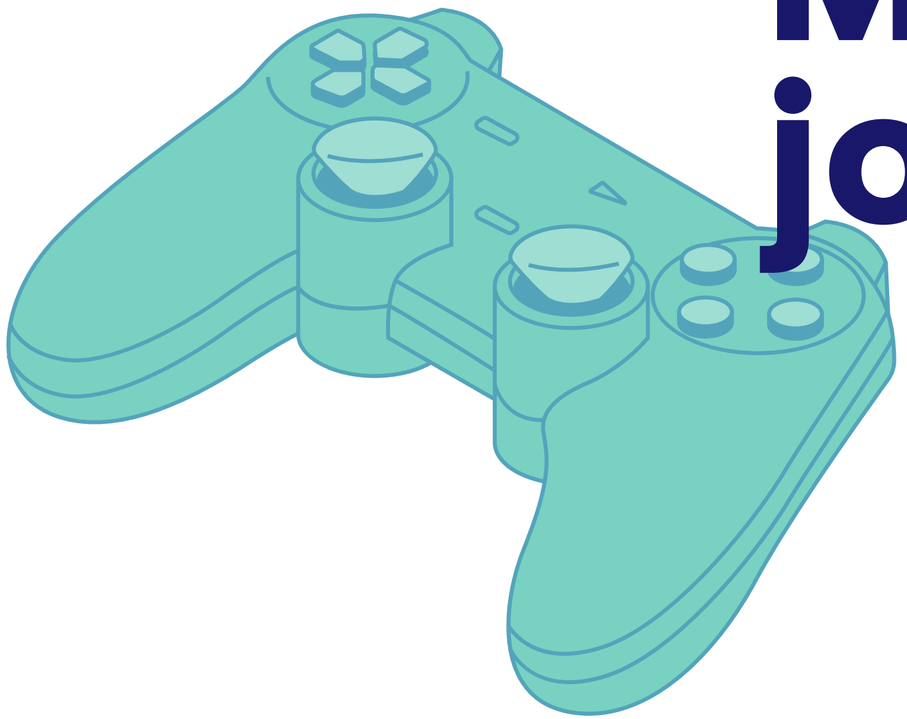
Maria Fernanda Mello

Como jogar?

Apenas pressione a alternativa que considere correta!

A cada rodada, o participante irá encarar perguntas e deve selecionar a alternativa correta. Lembre-se que o vencedor é aquele que acertar todas as respostas.

Insira seu nome, siga as instruções que aparecem na tela e se prepare!



Mecânica do jogo e regras:

- Se acertar cinco questões, leva 5000 pontos;
- Se acertar dez questões, garante 75000 pontos;
- Se acertar quinze questões, leva 250000 pontos;
- Se conseguir acertar todas as 20 questões, é o grande vencedor e leva 100000 de pontos!
- Em todas as perguntas existe a alternativa ‘E’, que permite que o jogador pule a questão sem ser eliminado,

Como rodar no seu computador?

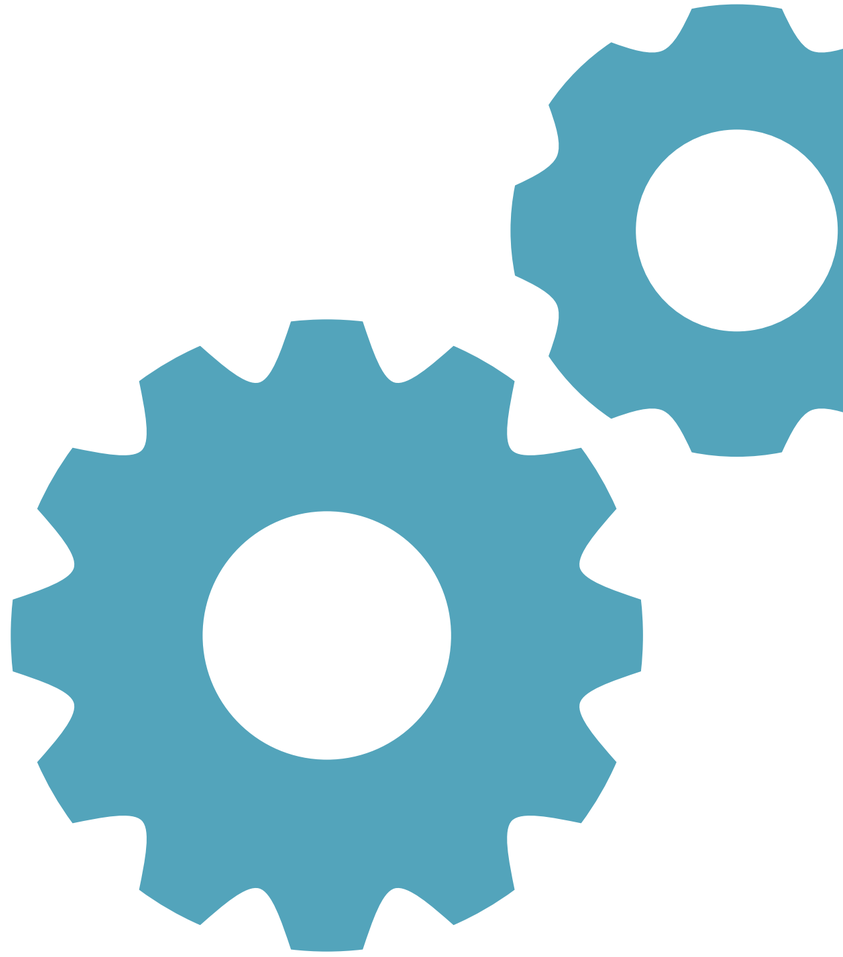
- Clone o repositório: https://github.com/mafemello/Operating-Systems-Game.git
- Abra o terminal e, na pasta em que o arquivo está, digite: make
- Após, digite: make run
- Pronto, o jogo irá começar, se prepare!

Implementação de Threads e Semáforos

Como nosso jogo funciona?

Display e Engine

- Thread para o display cuja função é gerenciar a impressão na tela do console
- Thread que se comporta como uma engine e lida com a maior parte da lógica do jogo, controlando o temporizador, a leitura das questões, pontuação e os restantes.
- Dois semáforos garantem a exclusão mútua do buffer permitindo a comunicação entre as threads citadas.



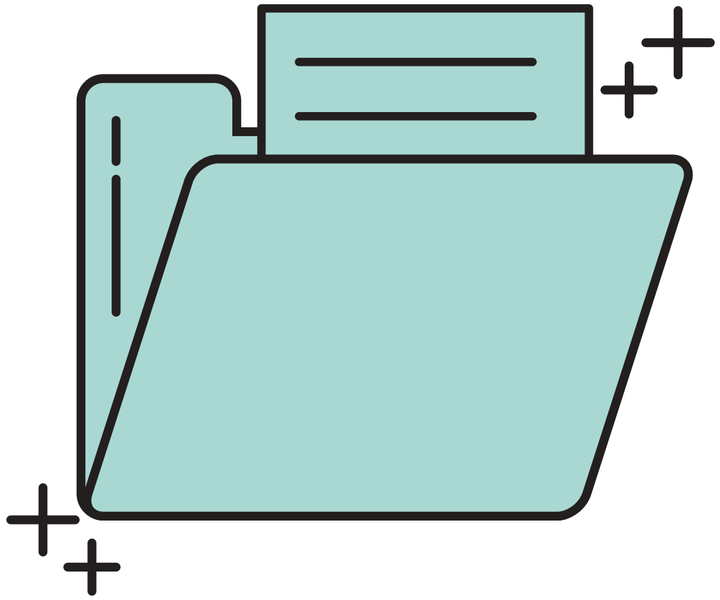
```cpp
Display::Display(SharedBuffer<DisplayContext> *buffer) {
    display_thread = std::thread([=]{
        PagesManager pages_manager(buffer);
        while(!pages_manager.is_at_last_page()) {
            pages_manager.display_page();
            pages_manager.next_page();
        }
        pages_manager.display_page();
    });
}

Display::~Display() {}

void Display::start_detached() {
    display_thread.detach();
}

void Display::start_joined() {
    display_thread.join();
}
```

```cpp
Engine::Engine(SharedBuffer<DisplayContext> *buffer, std::string questions_address)
        : context(questions_address, TIMER_TIMEOUT){
    current_controller = new StartingController(&context);
    engine_thread = std::thread([=] {
            while(!current_controller->is_last_controller()) {
                buffer->write(current_controller->get_display_context());
                current_controller->handle_input();
                delete_and_assign(&current_controller, current_controller->next());
            }
            buffer->write(current_controller->get_display_context());
            current_controller->handle_input();
    });
}

Engine::~Engine() {
    delete current_controller;
}

void Engine::start_detached() {
    engine_thread.detach();
}

void Engine::start_joined() {
    engine_thread.join();
}
```

```
template <typename T> class SharedBuffer {
    public:
        SharedBuffer() {
            sem_init(&can_read, 0, 0);
            sem_init(&can_write, 0, 0);
        }

        SharedBuffer(T content) {
            this->content = content;
            sem_init(&can_read, 0, 0);
            sem_init(&can_write, 0, 0);
        }

        T read() {
            allow_write();
            sem_wait(&can_read);
            return content;
        }

        void write(T content) {
            sem_wait(&can_write);
            this->content = content;
            allow_read();
        }

    private:
        sem_t can_read;
        sem_t can_write;
        T content;

        void allow_read() {
            sem_post(&can_read);
        }

        void allow_write() {
            sem_post(&can_write);
        }
};
```

Gerenciamento da leitura das questões

- Thread que gerencia a leitura do arquivo de questões;
- Para diminuir o tempo de leitura do arquivo, essa thread garante que quando uma questão é requisitada, ela já está na memória primária armazenada em um buffer que é compartilhado com a thread de engine.
- A implementação dessa dinâmica é feita através da classe QuestionsManager que possui SharedBuffer como dependência.



```
class QuestionsManager {
    private:
        sem_t stop;
        std::atomic<bool> is_running;
        std::ifstream questions_file;
        SharedBuffer<std::string> next_question_buffer;
        int current_question;

        void read_next_question();

    public:
        QuestionsManager(const std::string &questions_adderess);

        ~QuestionsManager();

        std::string get_next_question();

        bool answer_is_correct(char answer);

        bool answer_is_ignore(char answer);
};
```

```
void QuestionsManager::read_next_question() {
    if (is_running) {
        std::string next_question;
        std::getline(questions_file, next_question);
        std::string aux;
        for (int i = 0; i < NUMBER_OF_ALTERNATIVES; i++) {
            std::getline(questions_file, aux);
            next_question.append("\n").append(aux);
        }
        std::getline(questions_file, aux);
        next_question.append("\n").append(std::string(REFUSE_TO_ANSWER));
        next_question_buffer.write(next_question);
    }
}

QuestionsManager::QuestionsManager(const std::string &questions_address) {
    questions_file.open(questions_address);
    current_question = -1;
    is_running = true;
    sem_init(&stop, 0, 0);
    thread([=]{
        while(is_running) {
            read_next_question();
        }
        sem_post(&stop);
    }).detach();
}
```

```
std::string QuestionsManager::get_next_question() {
    if (current_question >= NUMBER_OF_QUESTIONS)
        return string("");
    std::string question = next_question_buffer.read();
    current_question++;
    return question;
}
```

Timer

- Temporizador Timer que faz uso de um semáforo para interromper a contagem regressiva do tempo;

```h
class Timer {
    private:
        sem_t stop;
        std::atomic<bool> timer_is_running;
        std::atomic<int> time_left;
        int starting_time;

    public:
        Timer(int starting_time);

        ~Timer();

        void start();

        void reset();

        std::string to_string();

        bool timed_out();

        bool is_running();
};
```

```cpp
Timer::Timer(int starting_time) {
    this->starting_time = starting_time;
    time_left = starting_time;
    timer_is_running = false;
    sem_init(&stop, 0, 0);
}

Timer::~Timer() {
    reset();
}

void Timer::start() {
    if (is_running() || timed_out())
        reset();
    timer_is_running = true;
    time_left = starting_time;
    std::thread([=]{
        while(time_left > 0 && timer_is_running) {
            std::this_thread::sleep_for(std::chrono::seconds(1));
            time_left--;
        }
        if (!timer_is_running)
            sem_post(&stop);
        timer_is_running = false;
    }).detach();
}

void Timer::reset() {
    if (timer_is_running) {
        timer_is_running = false;
        sem_wait(&stop);
        time_left = starting_time;
    }
}
```

Vamos jogar?