**Proposta de Threads e Suas Responsabilidades**

**Thread Principal (Main Thread)**

* **Responsabilidade:** A thread principal atuará como o orquestrador geral da aplicação. Suas funções primárias incluem:
  + Inicialização do ambiente do jogo: Configuração de bibliotecas gráficas ou de interface de texto (como ncurses, se utilizada para a interface ASCII mencionada na descrição do trabalho).
  + Criação e inicialização de todas as threads secundárias: thread\_helicoptero, thread\_bateria\_0, thread\_bateria\_1, e thread\_gerenciador\_jogo. As threads para os foguetes (thread\_foguete) serão criadas dinamicamente pelas threads das baterias.
  + Alocação e inicialização de todas as primitivas de sincronização (mutexes, variáveis de condição) e estruturas de dados compartilhadas que serão utilizadas pelas demais threads.
  + Após a criação das threads filhas persistentes, a thread principal aguardará a finalização de cada uma delas utilizando a chamada pthread\_join. Este passo é crucial, pois assegura que todas as threads completem suas tarefas e liberem seus recursos específicos antes que o programa principal prossiga para o encerramento. A não utilização de pthread\_join poderia levar a um término prematuro da thread principal, resultando na terminação abrupta das threads filhas e potencial vazamento de recursos ou estados inconsistentes.
  + Realização da limpeza final de todos os recursos globais: Destruição dos mutexes e variáveis de condição, liberação de memória alocada dinamicamente e encerramento formal do ambiente gráfico/texto.
  + Exibição da mensagem final de vitória ou derrota, conforme determinado pelo thread\_gerenciador\_jogo.
* **Justificativa:** Esta estrutura é padrão em aplicações multithreaded, centralizando o controle do ciclo de vida da aplicação e garantindo um gerenciamento de recursos robusto e ordenado, especialmente importante em cenários com múltiplas condições de término como as especificadas para o jogo.

**Thread do Helicóptero (Jogador) - thread\_helicoptero**

* **Responsabilidade:** Esta thread será dedicada a gerenciar todas as ações e o estado do helicóptero controlado pelo jogador. Suas tarefas incluem:
  + Processamento da entrada do jogador: Capturar e interpretar as teclas de seta (↑, ↓, ←, →) para determinar a intenção de movimento do helicóptero. Conforme a especificação, "O jogador pode movimentar o helicóptero para frente (seta →), para trás (seta ←), para cima (seta ↑) e para baixo (seta ↓)". Se nenhuma tecla for pressionada, o helicóptero deve parar.
  + Atualização da posição do helicóptero: Modificar as coordenadas do helicóptero na estrutura de dados compartilhada que representa o estado do jogo, respeitando os limites da tela.
  + Gerenciamento dos soldados: Controlar o número de soldados a bordo e o número de soldados já resgatados. Isso envolve simular o embarque dos 10 soldados na origem (esquerda da tela) e o desembarque na plataforma (direita da tela).
  + Detecção de colisões: Verificar continuamente se o helicóptero colidiu com:
    - Os limites da tela, especificamente o topo ("Se o helicóptero alcança o topo da tela (linha 0), ele explode").
    - Obstáculos no solo, incluindo "o próprio solo, alguma das baterias, o depósito ou a plataforma".
    - Foguetes disparados pelas baterias antiaéreas (lendo o estado dos foguetes ativos).
  + Atualização do estado do helicóptero: Modificar o estado do helicóptero (ex: ativo, explodido, missão cumprida) em uma estrutura de dados compartilhada, que será lida pela thread\_gerenciador\_jogo para determinar o fim do jogo.
* **Funcionamento Interno:** A lógica desta thread provavelmente envolverá um loop principal que, a cada iteração: (1) verifica a entrada do usuário (de forma não bloqueante para manter a responsividade); (2) atualiza a posição do helicóptero com base na entrada ou na ausência dela; (3) executa as rotinas de detecção de colisão com outros elementos do jogo (foguetes, baterias, cenário); e (4) atualiza o estado compartilhado do helicóptero. A interação precisa com dados de posição de outros objetos (baterias, foguetes) exigirá acesso sincronizado a esses dados.

**Thread da Bateria Antiaérea (2 instâncias) - thread\_bateria\_0 e thread\_bateria\_1**

* **Responsabilidade (para cada uma das duas instâncias):** Cada thread de bateria operará de forma independente em termos de sua lógica de comportamento, mas interagirá com recursos compartilhados (ponte, depósito) e com o estado do helicóptero. Propõe-se uma única função de thread que será parametrizada com um identificador (0 ou 1) e referências aos dados específicos de cada bateria. As responsabilidades incluem:
  + **Ciclo de Disparo:**
    - Verificar a quantidade de munição disponível (a bateria armazena n foguetes).
    - Se houver munição, disparar foguetes periodicamente "a esmo visando atingi-lo". Isso implica a necessidade de ler a posição atual do helicóptero (de forma sincronizada) para direcionar os disparos.
    - **Para cada foguete disparado, uma nova thread\_foguete será criada (ver seção 3.4).** A thread da bateria será responsável por iniciar esta nova thread, passando os parâmetros iniciais necessários (posição de origem, talvez um vetor de direção inicial).
    - Decrementar a contagem de munição após cada disparo.
  + **Ciclo de Recarga:**
    - Quando a munição se esgotar ("Quando os foguetes terminam"), a bateria deve iniciar o processo de recarga.
    - **Movimentação para o Depósito:** Deslocar-se da sua posição de combate para a esquerda, em direção à ponte.
    - **Travessia da Ponte (Ida):** Solicitar acesso exclusivo à ponte. Se a ponte estiver ocupada pela outra bateria, a thread deve aguardar. Após obter acesso, atravessar a ponte e, em seguida, liberar o acesso à ponte.
    - **Recarga no Depósito:** Chegando ao depósito (que é um recurso compartilhado, não uma thread separada de "recarregador"), a thread da bateria deve solicitar acesso para realizar a recarga. "Enquanto uma bateria está sendo recarregada, se a [outra] também necessita ser recarregada, ela deve aguardar." A sincronização para acesso ao depósito será detalhada na seção 5.2. A recarga tem duração variável ("de 1/10 s a 1/2 s"), dependente do nível de dificuldade. Após a recarga, a bateria libera o "uso" do depósito.
    - **Movimentação de Retorno:** Deslocar-se do depósito de volta para sua posição de combate original.
    - **Travessia da Ponte (Volta):** Novamente, solicitar acesso exclusivo à ponte, aguardar se necessário, atravessar e liberar.
  + Atualizar sua própria posição e estado (ex: ativa, recarregando, movendo-se para a ponte, na ponte, movendo-se para o depósito, etc.) em estruturas de dados compartilhadas para que outras threads (como a do helicóptero para colisões ou a do gerenciador de jogo para renderização) possam acessá-las.
* **Dados de Instância:** Cada bateria necessitará de sua própria estrutura de dados para armazenar sua posição, contagem de munição, estado atual.

**Thread do Foguete (Múltiplas Instâncias Dinâmicas) - thread\_foguete**

* **Responsabilidade:** Cada foguete disparado por uma bateria será gerenciado por sua própria thread individual. Esta thread será de curta duração.
  + **Criação:** Criada pela thread\_bateria\_N no momento do disparo.
  + **Movimentação:** Atualizar sua própria posição na tela de acordo com sua trajetória.
  + **Detecção de Limites/Colisão:** Verificar se atingiu os limites da tela ou se colidiu com o helicóptero. A detecção de colisão com o helicóptero pode ser uma responsabilidade primária desta thread ou da thread\_helicoptero. Se for desta thread, ela precisará ler a posição do helicóptero de forma sincronizada.
  + **Atualização de Estado:** Informar seu estado (ex: ativo, explodido, fora da tela) em uma estrutura de dados compartilhada (possivelmente uma lista de foguetes ativos, protegida por mutex).
  + **Término:** A thread termina quando o foguete sai da tela, atinge um alvo, ou é de alguma forma neutralizado. A criação e destruição frequente de threads pode introduzir overhead. Uma alternativa seria um pool de threads para foguetes, mas para simplificar, começaremos com criação/destruição dinâmica.
* **Justificativa:** Isolar a lógica de cada foguete em sua própria thread simplifica o gerenciamento de múltiplos projéteis simultâneos.

**Thread de Gerenciamento de Jogo/Estado Global - thread\_gerenciador\_jogo**

* **Responsabilidade:** Esta thread funcionará como o supervisor central do estado do jogo e da lógica de progressão. Suas funções são:
  + Monitoramento das Condições de Término: Verificar continuamente se alguma das três condições de término do jogo foi atingida:
    1. "o helicóptero de resgate cumprir a missão (vitória do jogador)" – 10 soldados resgatados.
    2. "o helicóptero é abatido (derrota do jogador)" – por um foguete.
    3. "o helicóptero se chica contra algum obstáculo (derrota do jogador)".
  + Sinalização de Fim de Jogo: Ao detectar uma condição de término, esta thread deve sinalizar para todas as outras threads (helicóptero, baterias, e possivelmente instruir as threads de foguete a não serem mais criadas ou a terminarem) que elas devem encerrar suas operações de forma controlada. Isso pode ser feito através de uma variável global booleana (protegida por mutex) indicando o fim do jogo.
  + Gerenciamento do Estado Global: Manter e atualizar informações globais como o nível de dificuldade selecionado ("o jogo tem 3 níveis de dificuldade"), pontuação (se implementada), e o número total de soldados resgatados (que é o critério para vitória).
  + **Coordenação da Atualização da Interface do Usuário (UI) / Renderização:** Esta thread será **exclusivamente** responsável por ler as posições atuais do helicóptero, das baterias, dos foguetes ativos, e o estado dos soldados a partir das estruturas de dados compartilhadas (acessando-as de forma sincronizada) e, em seguida, renderizar a tela do jogo. Para a "versão tosca da interface usando apenas caracteres ASCII", esta thread pode limpar a tela e redesenhar todos os elementos em suas posições atuais a cada ciclo de atualização. **Centralizar todas as operações de desenho (como gotoxy e escrita na tela) nesta única thread é crucial para garantir a atomicidade da renderização e evitar condições de corrida ou estado inconsistente da tela que ocorreriam se múltiplas threads tentassem modificar a UI concorrentemente**.
* **Benefícios da Centralização:** Ter uma thread dedicada ao gerenciamento do jogo simplifica a lógica de decisão de vitória/derrota. A centralização da renderização garante a integridade visual e evita problemas de concorrência com a biblioteca de UI (ex: ncurses, que não é inerentemente thread-safe ).

**Identificação de Condições de Corrida e Necessidades de Sincronização**

A natureza concorrente da aplicação, com múltiplas threads acessando e modificando dados compartilhados, introduz várias potenciais condições de corrida e, consequentemente, a necessidade de mecanismos de sincronização.

**Acesso à Ponte (Recurso Único Exclusivo)**

* **Recurso Compartilhado:** A ponte.
* **Condição de Corrida Potencial:** Ambas as thread\_bateria\_N tentando acessar/ocupar a ponte simultaneamente.
* **Necessidade de Sincronização:** Exclusão mútua para o uso da ponte.

**Acesso ao Depósito de Recarga (Recurso Único com Espera)**

* **Recurso Compartilhado:** O local físico/lógico do depósito onde ocorre a recarga. Este é um recurso passivo, não uma thread "recarregador" separada. A thread da bateria realiza a ação de recarregar *no* depósito.
* **Condição de Corrida Potencial e Ineficiência:** Uma bateria tentando recarregar enquanto outra já está no processo, ou espera ineficiente (busy-waiting).
* **Necessidade de Sincronização:** Exclusão mútua para o "ato de estar recarregando no depósito" e um mecanismo de espera eficiente para a bateria que aguarda a disponibilidade do depósito.

**Acesso e Modificação de Dados Globais do Jogo**

* **Recursos Compartilhados Críticos:**
  + Posição e Estado do Helicóptero.
  + Posições e Estados das Baterias.
  + **Lista/Array de Foguetes Ativos:** Se as thread\_foguete atualizam uma lista central de seus estados/posições (para leitura pela thread\_helicoptero para colisões e pela thread\_gerenciador\_jogo para renderização), esta lista é um recurso compartilhado crítico. Adicionar/remover foguetes ou atualizar seus dados nesta lista precisa ser sincronizado.
  + Estado dos Soldados.
  + Estado Geral do Jogo (flag jogo\_em\_andamento, etc.).
  + Contagem de Munição das Baterias.
* **Condição de Corrida Potencial:** Leituras inconsistentes, atualizações perdidas, corrupção de estruturas de dados (como a lista de foguetes).
* **Necessidade de Sincronização:** Exclusão mútua para todas essas estruturas.

**Detecção de Colisões**

* **Interação:** A thread\_helicoptero precisa verificar colisões com foguetes (cujos dados podem estar em uma lista compartilhada atualizada pelas thread\_foguete) e com as próprias baterias. As thread\_foguete podem também verificar colisão com o helicóptero.
* **Condição de Corrida Potencial:** Ler dados de posição de objetos que estão sendo movidos concorrentemente por outras threads, levando a detecções de colisão imprecisas.
* **Necessidade de Sincronização:** Acesso sincronizado aos dados de posição dos objetos envolvidos na verificação de colisão.

**Renderização da Interface do Usuário**

* **Recurso Compartilhado:** A tela/terminal.
* **Condição de Corrida Potencial:** Se múltiplas threads tentassem chamar funções de desenho (ex: gotoxy, printw em ncurses) simultaneamente, poderia ocorrer sobreposição de escrita, tela corrompida, ou comportamento indefinido da biblioteca de UI, já que muitas não são thread-safe.
* **Necessidade de Sincronização:** Garantir que apenas uma thread (thread\_gerenciador\_jogo) realize operações de desenho. Isso é uma forma de serialização do acesso ao recurso "tela".

**Estratégias de Tratamento (Sincronização com pthreads)**

Para tratar as condições de corrida e necessidades de sincronização identificadas, propomos o uso dos seguintes mecanismos da biblioteca pthreads:

**Para Acesso à Ponte**

* **Mecanismo:** Um único mutex global pthread\_mutex\_t, mutex\_ponte.
* **Utilização:** Bloquear antes de usar, desbloquear depois.

**Para Acesso ao Depósito de Recarga**

* **Mecanismos:** Um pthread\_mutex\_t (mutex\_deposito), uma pthread\_cond\_t (cond\_deposito\_livre), e uma variável booleana (deposito\_ocupado) protegida por mutex\_deposito.
* **Utilização:** A thread\_bateria\_N que deseja recarregar trava mutex\_deposito. Se deposito\_ocupado for true, ela chama pthread\_cond\_wait em cond\_deposito\_livre (liberando mutex\_deposito e esperando). Quando a outra bateria termina a recarga, ela define deposito\_ocupado para false (sob o mutex\_deposito) e sinaliza cond\_deposito\_livre com pthread\_cond\_signal (ou pthread\_cond\_broadcast), acordando a thread em espera. A thread acordada re-verifica a condição (pois pthread\_cond\_wait re-adquire o mutex).
* **Clarificação:** A "interação entre bateria e recarregador" é, na verdade, a interação da thread da bateria com o *recurso compartilhado* "depósito". Não há uma "thread recarregador" separada; a própria thread da bateria executa a lógica de recarga após obter acesso exclusivo ao local/facilidade de recarga (o depósito).

**Para Atualização e Leitura de Dados Globais e de Entidades do Jogo**

* **Mecanismo:** Múltiplos mutexes (pthread\_mutex\_t) com granularidade apropriada:
  + mutex\_helicoptero\_dados
  + mutex\_bateria\_0\_dados
  + mutex\_bateria\_1\_dados
  + **mutex\_lista\_foguetes**: Para proteger a lista/array de foguetes ativos, se esta abordagem for usada para centralizar os dados dos foguetes. As thread\_foguete travariam este mutex para adicionar/remover a si mesmas ou atualizar seu estado, e a thread\_helicoptero e thread\_gerenciador\_jogo travariam para ler.
  + mutex\_estado\_jogo
* **Utilização:** Adquirir lock antes de acessar/modificar, liberar depois. Manter uma ordem de travamento consistente para evitar deadlocks.

**Para Detecção de Colisões**

* **Mecanismo:** Utilizar os mutexes de dados das entidades envolvidas (mutex\_helicoptero\_dados, mutex\_lista\_foguetes ou mutexes individuais de foguetes se não houver lista central, mutex\_bateria\_N\_dados), respeitando a ordem de travamento.
* **Utilização:** Adquirir os locks necessários, ler posições, calcular, liberar locks.

**Para Renderização da Interface do Usuário**

* **Mecanismo:** Nenhuma primitiva de sincronização explícita é necessária *entre threads* para o ato de desenhar, pois a responsabilidade de todas as operações de UI será **exclusivamente** da thread\_gerenciador\_jogo.
* **Utilização:** A thread\_gerenciador\_jogo lerá os dados de estado das outras entidades (helicóptero, baterias, foguetes) de forma sincronizada (usando os respectivos mutexes como mutex\_helicoptero\_dados, mutex\_lista\_foguetes, etc.). Após coletar todas as informações necessárias, ela realizará as operações de desenho na tela. Como nenhuma outra thread tentará desenhar, não haverá conflito direto nas chamadas da biblioteca de UI. Isso garante a atomicidade da atualização da tela em relação às outras lógicas do jogo.

**Tabela Resumo: Threads e Mecanismos de Sincronização**

| Thread | Responsabilidade Principal | Recursos Compartilhados Chave Acessados | Mecanismos pthreads Propostos |
| --- | --- | --- | --- |
| Principal (Main) | Inicialização, criação/join de threads persistentes, limpeza | Estruturas de sincronização (criação/destruição) | pthread\_create, pthread\_join |
| Helicóptero (Jogador) | Movimentação, gestão de soldados, detecção de colisão | Posição/estado do helicóptero, estado dos soldados, posições de baterias/foguetes (leitura da lista de foguetes), estado global do jogo | mutex\_helicoptero\_dados, mutex\_estado\_jogo, mutex\_lista\_foguetes (leitura) |
| Bateria Antiaérea (0 e 1) | Disparar (criar thread\_foguete), mover para recarga, atravessar ponte, recarregar no depósito | Ponte, Depósito, Posição/estado da bateria, munição, Posição do helicóptero (leitura), estado global do jogo | mutex\_ponte, mutex\_deposito + cond\_deposito\_livre, mutex\_bateria\_N\_dados, (leitura de mutex\_helicoptero\_dados) |
| Foguete (Múltiplas) | Movimentação, autodestruição (limites/colisão), atualizar estado na lista | Sua própria posição/estado, lista de foguetes ativos (escrita), posição do helicóptero (leitura para colisão) | mutex\_lista\_foguetes (escrita/atualização), (leitura de mutex\_helicoptero\_dados) |
| Gerenciamento de Jogo/Estado | Monitorar fim de jogo, gerenciar estado global, **renderizar UI exclusivamente** | Estado global do jogo, posições de todos os elementos (leitura para renderização: helicóptero, baterias, lista de foguetes), contagem de soldados resgatados | mutex\_estado\_jogo, (leitura de mutex\_helicoptero\_dados, mutex\_bateria\_N\_dados, mutex\_lista\_foguetes para renderização) |