Algoritmo Hierarchical Timing Wheel: Uma Abordagem para Gerenciamento Eficiente de Temporizadores

Introdução e Histórico

O algoritmo **Hierarchical Timing Wheel** foi introduzido para resolver problemas de eficiência no gerenciamento de temporizadores em sistemas computacionais, especialmente em sistemas operacionais e redes de computadores.

Originalmente, o problema central era: **como gerenciar um grande número de temporizadores com eficiência de tempo e espaço?** Antes do surgimento dos timing wheels, as abordagens mais comuns usavam estruturas de dados como listas encadeadas ordenadas ou filas de prioridade (heaps), que, apesar de funcionais, tinham limitações significativas de desempenho.

Um marco importante é o trabalho de Varghese e Lauck (1996), que popularizaram o uso dos **Hierarchical Timing Wheels** para escalonar temporizadores em sistemas operacionais de forma eficiente:

> "The hierarchical timing wheel provides an O(1) amortized complexity for timer management, overcoming limitations of simple lists or priority queues."
> — *Varghese and Lauck, 1996*

Contexto: O Que Era Feito Antes?

Antes do Hierarchical Timing Wheel, as técnicas mais comuns para o gerenciamento de temporizadores incluíam:

- **Listas Encadeadas Ordenadas:** Temporizadores eram armazenados em uma lista, ordenada pelo tempo de disparo. A complexidade para inserção era O(n), e para avanço do tempo, era necessário percorrer a lista para identificar os timers a disparar.
- **Filas de Prioridade (Heaps):** Usavam estruturas como heaps binários, que garantem inserção e remoção do menor elemento em O(log n). Embora melhor que listas ordenadas, o custo ainda era alto para sistemas com milhares de timers.

Essas abordagens apresentavam **problemas de escalabilidade** em sistemas de alta performance, como servidores web, roteadores e sistemas operacionais modernos.

Por que o Hierarchical Timing Wheel Surgiu?

O objetivo era criar um algoritmo que possibilitasse:

- **Operações em tempo constante amortizado (O(1))**, para inserção, remoção e disparo de temporizadores.
- **Baixo custo computacional mesmo com um grande número de temporizadores**.

- **Eficiência espacial**, minimizando a memória usada para manter os timers.

O conceito básico do timing wheel é inspirado no funcionamento de um relógio analógico, onde o ponteiro avança e executa ações em tempos específicos, representando temporizadores como entradas em “slots” do círculo.

Funcionamento do Hierarchical Timing Wheel

O Hierarchical Timing Wheel é uma estrutura de múltiplos níveis, onde cada nível representa uma faixa de tempo maior que o anterior, formando uma hierarquia. Cada nível é um “anel” com um número fixo de slots (como os minutos e horas em um relógio).

- Temporizadores com duração curta são armazenados no nível mais baixo.
- Temporizadores com durações maiores são “promovidos” a níveis superiores, que avançam mais lentamente.
- Quando um nível mais baixo completa um ciclo, os temporizadores são promovidos para o próximo nível, semelhante à “transição” dos minutos para as horas.

Essa estrutura permite que a busca e o disparo dos temporizadores ocorram em tempo constante, pois a operação se restringe a um slot do anel.

Vantagens

1. **Complexidade O(1) amortizada:** Como mostrado por Varghese e Lauck (1996), as operações básicas não dependem do número total de timers, tornando o sistema altamente escalável.
2. **Eficiência espacial:** Ao usar uma hierarquia de rodas, o algoritmo evita alocar espaço excessivo para tempos muito longos.
3. **Simplicidade de implementação:** Conceitualmente, é mais simples do que estruturas complexas como heaps binários, e se integra bem em kernels e sistemas em tempo real.
4. **Baixa latência:** O acesso direto aos slots evita buscas longas e garante que timers sejam disparados pontualmente.

Desvantagens

1. **Granularidade limitada:** A resolução dos temporizadores é limitada ao tamanho do slot mais baixo. Para aplicações que requerem alta precisão, o timing wheel pode ser insuficiente.
2. **Sobrecarga em caso de timers com duração muito longa:** O algoritmo pode gastar tempo promovendo timers entre níveis da hierarquia.
3. **Complexidade de ajuste de parâmetros:** O número de níveis e tamanho dos slots precisam ser cuidadosamente escolhidos para equilibrar eficiência e precisão.
4. **Overhead em sistemas com muitos timers expostos a modificações frequentes:** Alterações constantes podem levar a movimentações complexas na hierarquia.

Aplicações Relevantes

O Hierarchical Timing Wheel é amplamente usado em:

- Sistemas operacionais para gerenciamento de timers de processos.
- Redes de computadores para controle de timeouts e retransmissões.
- Jogos e simulações em tempo real, onde muitos eventos temporizados são comuns.

Claro! Vou continuar o texto da aula explicando o algoritmo do **Hierarchical Timing Wheel** de forma matemática e detalhada, mantendo o foco acadêmico e técnico.

Detalhes Matemáticos do Algoritmo Hierarchical Timing Wheel

Definição Formal

Considere um sistema onde temos um conjunto de temporizadores $T = {t_1, t_2, \ldots, t_n}$, cada um associado a um tempo de expiração e_i, medido em “ticks” (unidade discreta de tempo, como milissegundos).

O problema é: dado o tempo atual c, devemos disparar todos os temporizadores t_i tal que $e_i \leq c$, e garantir que operações de inserção, remoção e avanço do tempo sejam eficientes.

Estrutura do Timing Wheel

O Hierarchical Timing Wheel consiste em uma hierarquia de L níveis, onde cada nível l é um array circular (ou “roda”) de N_l slots. Cada slot corresponde a um intervalo de tempo fixo, chamado de resolução daquele nível.

Definimos:

- R_0: resolução do nível 0 (mais baixo), i.e., duração de um slot no nível 0 (ex: 1 ms)
- N_l: número de slots no nível l
- A resolução do nível l é dada por:

$$
R_l = R_0 \times \prod_{i=0}^{l-1} N_i
$$

Ou seja, o nível 1 tem resolução $R_1 = R_0 \times N_0$, nível 2 tem $R_2 = R_1 \times N_1 = R_0 \times N_0 \times N_1$, e assim por diante.

Representação do Tempo e Mapeamento para Slots

Para um temporizador com tempo de expiração e, queremos colocá-lo no nível l e slot s apropriados.

Definimos:

$$
\Delta = e - c
$$

onde Δ é a diferença entre o tempo de expiração e o tempo atual.

O algoritmo decide o nível l de acordo com o menor l para o qual:

$$
\Delta < R_{l+1} = R_0 \times \prod_{i=0}^{l} N_i
$$

Ou seja, o temporizador cabe no intervalo coberto pelo nível l.

Uma vez escolhido o nível l, o slot s onde o temporizador será inserido é:

$$
s = \left\lfloor \frac{e}{R_l} \right\rfloor \bmod N_l
$$

O operador módulo garante a circularidade da roda.

Exemplo

Suponha:

- $R_0 = 1$ ms
- $N_0 = 256$ slots
- $N_1 = 64$ slots

Então,

- $R_1 = 1 \times 256 = 256$ ms
- $R_2 = 256 \times 64 = 16384$ ms

Para um temporizador com $\Delta = 500$ ms, temos:

- $\Delta = 500$ ms
- $\Delta < R_2 = 16384$, mas $\Delta > R_1 = 256$, logo $l = 1$.

Slot será:

$$
s = \left\lfloor \frac{c + 500}{256} \right\rfloor \bmod 64
$$

Operação de Avanço do Tempo (Tick)

A cada unidade de tempo R_0 (1 ms no exemplo), o ponteiro do nível 0 avança 1 slot.

Quando o ponteiro de um nível l completa uma volta (i.e., avança de slot 0 a $N_l - 1$), ocorre um **overflow** e os temporizadores no slot corrente daquele nível são promovidos para o nível $l-1$.

Formalmente:

- Para cada nível l, o ponteiro p_l indica o slot atual.

- Quando p_l avança e completa uma volta, os temporizadores no slot p_l do nível l são redistribuídos para slots adequados no nível $l-1$ de acordo com o cálculo do slot citado acima.

Complexidade Temporal

- **Inserção:** A escolha do nível e cálculo do slot é feita em tempo constante $O(1)$, com base na comparação de Δ com R_l e cálculo direto do slot.
- **Avanço do ponteiro (tick):** Normalmente, movimenta o ponteiro de nível 0 e dispara timers no slot atual. Quando ocorre overflow, há redistribuição dos timers, que é amortizada em relação ao número total de timers.

Por isso, a complexidade amortizada por operação é $O(1)$.

Espaço

O espaço utilizado é $O(\sum_{l=0}^{L-1} N_l)$, proporcional ao número total de slots na hierarquia, mais o espaço necessário para armazenar temporizadores em listas vinculadas dentro dos slots.

Análise Matemática da Eficiência

A eficiência do algoritmo deriva da segmentação do tempo em múltiplos níveis, onde cada nível gerencia um conjunto distinto de temporizadores em uma faixa de tempo específica.

Isso evita buscas lineares (O(n)) e ordenações (O($n \log n$)) das técnicas anteriores.

O tempo médio por operação é:

$$
T_{op} = \frac{1}{n} \sum_{i=1}^n t_i
$$

onde t_i é o tempo gasto na inserção, remoção ou disparo de cada temporizador.

Devido à estrutura hierárquica e redistribuição controlada, T_{op} permanece constante, conforme demonstrado em:

> "The amortized cost per timer operation is $O(1)$, making hierarchical timing wheels suitable for high-performance timer management."
> — *Varghese and Lauck, 1996*

Matematicamente, o Hierarchical Timing Wheel é uma estrutura que mapeia tempos discretos em slots de anéis circulares, distribuídos hierarquicamente para manter eficiência e escalabilidade.

A formulação em termos de:

- níveis hierárquicos,
- resolução crescente R_l,
- cálculo modular do slot,

permite implementar um sistema de temporizadores com garantias claras de complexidade e performance.

Por que usar o algoritmo Hierarchical Timing Wheel?

1. O que é um temporizador em computação?

Um **temporizador** é um mecanismo que permite que um programa ou sistema execute uma ação após um determinado intervalo de tempo. Por exemplo:

- Disparar um evento após 5 segundos.
- Enviar um sinal de timeout se uma operação demora demais.
- Realizar uma ação periódica a cada 100 ms.

Em termos práticos, temporizadores são contadores que "contam o tempo" e avisam o sistema quando o tempo configurado acabou.

---

2. Por que os temporizadores são importantes?

Temporizadores são essenciais para:

- **Gerenciamento de recursos:** Um sistema operacional usa temporizadores para escalonar processos, garantindo que nenhum processo monopolize o CPU.
- **Comunicação em redes:** Protocolos como TCP usam temporizadores para retransmitir pacotes perdidos ou detectar falhas.
- **Sistemas em tempo real:** Aplicações que precisam responder dentro de prazos específicos, como controle industrial e jogos.
- **Aplicações gerais:** Em interfaces gráficas, para animações ou para sincronizar tarefas.

Sem temporizadores, seria impossível fazer controle eficiente do tempo em sistemas computacionais.

---

3. Onde são usados temporizadores?

Temporizadores são usados em diversos contextos, por exemplo:

- **Sistemas operacionais:** escalonamento, gerenciamento de processos, detecção de deadlocks.
- **Redes:** timeouts, retransmissões, detecção de pacotes atrasados.
- **Aplicações embarcadas:** sensores, atuadores, controle de dispositivos.
- **Jogos e simulações:** para eventos periódicos e controle de animações.
- **Sistemas distribuídos:** sincronização e coordenação entre nós.

---

4. O que acontece se não usarmos temporizadores de forma eficiente?

Se os temporizadores não forem gerenciados adequadamente, podem ocorrer vários problemas:

- **Desempenho degradado:** Algoritmos ineficientes para gerenciar temporizadores podem consumir muito CPU, causando lentidão.

- **Falhas em tempo real:** Sistemas podem não responder no tempo esperado, comprometendo a confiabilidade.
- **Uso excessivo de memória:** Estruturas de dados mal projetadas podem ocupar muita memória, inviabilizando o sistema.
- **Inconsistência na execução:** Temporizadores podem disparar com atraso ou antecipadamente, causando erros.

Por isso, **o gerenciamento eficiente dos temporizadores é crítico para o desempenho e a estabilidade dos sistemas.**

---

5. Por que usar o Hierarchical Timing Wheel para gerenciar temporizadores?

Como vimos, gerenciar temporizadores pode ser custoso se usarmos listas simples ou filas de prioridade, especialmente quando temos milhares ou milhões deles.

O **Hierarchical Timing Wheel** surgiu para:

- **Reduzir o custo das operações** de inserção, remoção e disparo para tempo constante $O(1)$ amortizado.
- **Escalar para muitos temporizadores** sem degradar o desempenho.
- **Minimizar overhead de CPU e memória**, ao organizar os temporizadores em múltiplos níveis com resoluções variadas.
- **Garantir disparo pontual e previsível** dos temporizadores, essencial para sistemas em tempo real.

Em resumo, o algoritmo é ideal quando:

- Existem muitos temporizadores ativos.
- O sistema precisa responder rapidamente a disparos.
- A performance é crítica e a eficiência do gerenciamento impacta diretamente o funcionamento.

---

6. E se eu usar uma estrutura simples, como uma lista ordenada?

- Inserção de temporizadores custa $O(n)$ — lento se muitos temporizadores.
- Avanço do tempo exige percorrer a lista para encontrar timers que expiraram — pode causar atrasos.
- Uso de CPU e memória aumenta com a quantidade de temporizadores.
- Pode causar falhas em sistemas sensíveis a tempo.

---

7. Em resumo, o Hierarchical Timing Wheel resolve:

- Como organizar temporizadores para que inserções e remoções sejam rápidas.
- Como disparar temporizadores exatamente no tempo esperado.
- Como manter desempenho constante mesmo com muitos temporizadores.

Aplicações Práticas do Hierarchical Timing Wheel no Dia a Dia

Onde o Hierarchical Timing Wheel é usado no cotidiano?

Apesar de ser um algoritmo mais técnico, o Hierarchical Timing Wheel está presente em muitos sistemas que usamos diariamente, ainda que de forma invisível para o usuário final. Ele é fundamental para garantir que temporizadores e eventos temporais sejam gerenciados com alta eficiência em sistemas complexos.

Exemplos comuns de uso

1. **Sistemas Operacionais**

O kernel de sistemas operacionais como Linux ou Windows precisa gerenciar milhares de temporizadores para escalonamento de processos, timeouts de I/O, timers de rede e outros. O Hierarchical Timing Wheel ou variações dele são usadas para garantir que o sistema responda rapidamente e gerencie o tempo eficientemente.

2. **Redes de Computadores**

Protocolos de comunicação, como TCP/IP, dependem de temporizadores para retransmissão de pacotes, manutenção de conexões e detecção de falhas. Grandes servidores de internet, roteadores e firewalls usam esse algoritmo para controlar centenas de milhares de temporizadores simultâneos com baixo overhead.

3. **Jogos e Simulações**

Em jogos online e motores de simulação, eventos como animações, habilidades com cooldown, e spawn de inimigos precisam ser ativados no tempo certo. O uso do Hierarchical Timing Wheel ajuda a garantir que esses eventos sejam disparados pontualmente, mesmo quando muitos temporizadores estão ativos.

4. **Sistemas Embarcados e IoT**

Dispositivos embarcados, como sensores e atuadores conectados na Internet das Coisas, precisam de temporizadores para monitoramento, controle e sincronização. O algoritmo ajuda a gerenciar esses eventos com baixo consumo de recursos computacionais.

Como aplicar o Hierarchical Timing Wheel em projetos?

Se você estiver desenvolvendo um projeto que precise lidar com muitos temporizadores simultâneos, especialmente quando a precisão e o desempenho são importantes, considere usar esse algoritmo.

Passos para aplicar:

1. **Avalie o requisito de temporização**

   Determine o número de temporizadores, a faixa de tempos que precisam ser cobertos, e a precisão desejada.

2. **Configure os níveis e resoluções**

Defina quantos níveis a hierarquia terá, quantos slots cada nível terá e qual será a resolução de tempo para cada nível. Isso depende da faixa de tempos e da granularidade desejada.

3. **Implemente as operações básicas**
    - Inserção: calcular o nível e slot baseado no tempo de expiração.
    - Avanço do tempo: movimentar ponteiros e promover temporizadores entre níveis.
    - Disparo: executar a ação do temporizador quando expirar.
4. **Teste com cargas reais**

    Verifique a performance com o número esperado de temporizadores e ajuste parâmetros (número de slots, níveis) para otimizar.
5. **Integre ao sistema**

    Use o Hierarchical Timing Wheel como componente do gerenciador de eventos ou scheduler da aplicação.

Exemplo prático simples (pseudocódigo):

```
class TimingWheel:
    def __init__(self, resolution, slots, levels):
        self.resolution = resolution  # base time slot (e.g., 1 ms)
        self.slots = slots            # number of slots per level
        self.levels = levels          # number of hierarchical levels
        self.wheels = [[[] for _ in range(slots)] for _ in
range(levels)]
        self.current_pos = [0]*levels
        self.current_time = 0

    def add_timer(self, expiration, callback):
        delta = expiration - self.current_time
        level = self.find_level(delta)
        slot = (expiration // (self.resolution * (self.slots ** level)))
% self.slots
        self.wheels[level][slot].append(callback)

    def tick(self):
        # Move pointer on level 0
        pos = self.current_pos[0]
        timers = self.wheels[0][pos]
        for cb in timers:
            cb()
        self.wheels[0][pos] = []
        self.current_pos[0] = (pos + 1) % self.slots
        self.current_time += self.resolution
        # Handle promotions to lower levels if necessary
```

O Hierarchical Timing Wheel é uma ferramenta poderosa para projetos que precisam gerenciar muitos temporizadores de forma eficiente e com baixa latência. Ele está presente em sistemas críticos do dia a dia, como sistemas operacionais, redes, jogos e dispositivos conectados.

Se o seu projeto envolve muitas operações temporizadas e exige alta performance, investir na implementação ou uso de bibliotecas baseadas nesse algoritmo pode garantir escalabilidade e robustez.

Implementação Simples do Hierarchical Timing Wheel

1. Python

```
import time

class TimingWheel:
    def __init__(self, resolution_ms=100, slots=256, levels=4):
        self.resolution = resolution_ms  # em milissegundos
        self.slots = slots
        self.levels = levels
        # Criar uma roda hierárquica: lista de níveis, cada um com uma
lista de slots (listas)
        self.wheels = [[[] for _ in range(slots)] for _ in
range(levels)]
        self.current_pos = [0] * levels
        self.current_time = 0  # contador do tempo em milissegundos

    def _find_level(self, delta_ms):
        # Determina em qual nível colocar o temporizador, baseado no
delta (intervalo)
        max_span = self.resolution
        for level in range(self.levels):
            if delta_ms < max_span * self.slots:
                return level
            max_span *= self.slots
        return self.levels - 1

    def add_timer(self, timeout_ms, callback):
        expiration = self.current_time + timeout_ms
        delta = expiration - self.current_time
        level = self._find_level(delta)
        slot_span = self.resolution * (self.slots ** level)
        slot = (expiration // slot_span) % self.slots
        self.wheels[level][int(slot)].append((expiration, callback))
        print(f"Timer adicionado no nível {level}, slot {int(slot)},
para {timeout_ms} ms.")

    def _cascade(self, level):
```

```
        pos = self.current_pos[level]
        timers = self.wheels[level][pos]
        self.wheels[level][pos] = []

        for expiration, callback in timers:
            if expiration <= self.current_time:
                callback()
            else:
                # Reinsere no nível inferior
                if level > 0:
                    self._add_timer_at_level(expiration, callback, level
- 1)
                else:
                    # Caso no nível 0 e não expirou, re-coloca no slot
correto
                    self.add_timer(expiration - self.current_time,
callback)

    def _add_timer_at_level(self, expiration, callback, level):
        slot_span = self.resolution * (self.slots ** level)
        slot = (expiration // slot_span) % self.slots
        self.wheels[level][int(slot)].append((expiration, callback))

    def tick(self):
        # Avança o ponteiro no nível 0
        self.current_time += self.resolution
        pos = self.current_pos[0]
        timers = self.wheels[0][pos]
        self.wheels[0][pos] = []

        for expiration, callback in timers:
            if expiration <= self.current_time:
                callback()
            else:
                # Se não expirou, re-insere
                self.add_timer(expiration - self.current_time, callback)

        self.current_pos[0] = (pos + 1) % self.slots

        # Cascata para níveis superiores se ponteiro zerar
        for level in range(1, self.levels):
            if self.current_pos[level - 1] == 0:
                pos_level = self.current_pos[level]
                self._cascade(level)
                self.current_pos[level] = (pos_level + 1) % self.slots

# Teste básico

def my_callback():
    print(f"Timer disparado no tempo {int(time.time() * 1000)} ms")

wheel = TimingWheel(resolution_ms=100, slots=64, levels=3)
```

```
wheel.add_timer(150, my_callback)   # dispara depois de 150ms
wheel.add_timer(700, my_callback)   # dispara depois de 700ms
wheel.add_timer(5000, my_callback)  # dispara depois de 5s

start = time.time()
while time.time() - start < 6:
    wheel.tick()
    time.sleep(0.1)
```

---

2. C (versão simplificada)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>

#define SLOTS 64
#define LEVELS 3
#define RESOLUTION_MS 100

typedef void (*callback_t)(void);

typedef struct Timer {
    uint64_t expiration;
    callback_t callback;
    struct Timer* next;
} Timer;

typedef struct {
    Timer* slots[SLOTS];
    int current_pos;
} WheelLevel;

typedef struct {
    WheelLevel levels[LEVELS];
    uint64_t current_time;
} TimingWheel;

// Função para criar timer
Timer* create_timer(uint64_t expiration, callback_t cb) {
    Timer* t = (Timer*)malloc(sizeof(Timer));
    t->expiration = expiration;
    t->callback = cb;
    t->next = NULL;
    return t;
}

void add_timer_at_level(TimingWheel* tw, uint64_t expiration, callback_t
cb, int level);
```

```c
int find_level(uint64_t delta) {
    uint64_t max_span = RESOLUTION_MS;
    for (int i = 0; i < LEVELS; i++) {
        if (delta < max_span * SLOTS)
            return i;
        max_span *= SLOTS;
    }
    return LEVELS - 1;
}

void add_timer(TimingWheel* tw, uint64_t timeout, callback_t cb) {
    uint64_t expiration = tw->current_time + timeout;
    uint64_t delta = expiration - tw->current_time;
    int level = find_level(delta);

    uint64_t slot_span = RESOLUTION_MS;
    for (int i = 0; i < level; i++) slot_span *= SLOTS;

    int slot = (expiration / slot_span) % SLOTS;

    Timer* timer = create_timer(expiration, cb);
    Timer** slot_list = &tw->levels[level].slots[slot];

    // Inserção no início da lista ligada
    timer->next = *slot_list;
    *slot_list = timer;

    printf("Timer adicionado no nível %d, slot %d, para %llu ms.\n",
level, slot, timeout);
}

void execute_timers(TimingWheel* tw, int level, int slot) {
    Timer* curr = tw->levels[level].slots[slot];
    Timer* prev = NULL;
    Timer* next = NULL;

    while (curr != NULL) {
        next = curr->next;
        if (curr->expiration <= tw->current_time) {
            curr->callback();
            free(curr);
            if (prev == NULL) {
                tw->levels[level].slots[slot] = next;
            } else {
                prev->next = next;
            }
        } else {
            prev = curr;
        }
        curr = next;
    }
}
```

```
void cascade(TimingWheel* tw, int level) {
    int pos = tw->levels[level].current_pos;
    Timer* timers = tw->levels[level].slots[pos];
    tw->levels[level].slots[pos] = NULL;

    Timer* curr = timers;
    while (curr != NULL) {
        Timer* next = curr->next;
        if (curr->expiration <= tw->current_time) {
            curr->callback();
            free(curr);
        } else {
            if (level > 0)
                add_timer_at_level(tw, curr->expiration, curr->callback,
level - 1);
            else
                add_timer(tw, curr->expiration - tw->current_time, curr-
>callback);
            free(curr);
        }
        curr = next;
    }
}

void add_timer_at_level(TimingWheel* tw, uint64_t expiration, callback_t
cb, int level) {
    uint64_t slot_span = RESOLUTION_MS;
    for (int i = 0; i < level; i++) slot_span *= SLOTS;

    int slot = (expiration / slot_span) % SLOTS;

    Timer* timer = create_timer(expiration, cb);
    Timer** slot_list = &tw->levels[level].slots[slot];

    timer->next = *slot_list;
    *slot_list = timer;
}

void tick(TimingWheel* tw) {
    tw->current_time += RESOLUTION_MS;

    // Nível 0
    int pos0 = tw->levels[0].current_pos;
    execute_timers(tw, 0, pos0);
    tw->levels[0].current_pos = (pos0 + 1) % SLOTS;

    // Cascata nos níveis superiores
    for (int i = 1; i < LEVELS; i++) {
        if (tw->levels[i - 1].current_pos == 0) {
            cascade(tw, i);
            tw->levels[i].current_pos = (tw->levels[i].current_pos + 1)
% SLOTS;
```

```
        }
    }
}

// Callback de exemplo
void my_callback() {
    printf("Timer disparado no tempo %llu ms\n", (unsigned long
long)clock());
}

int main() {
    TimingWheel tw = {0};
    for (int i = 0; i < LEVELS; i++) {
        tw.levels[i].current_pos = 0;
        for (int j = 0; j < SLOTS; j++)
            tw.levels[i].slots[j] = NULL;
    }
    tw.current_time = 0;

    add_timer(&tw, 150, my_callback);
    add_timer(&tw, 700, my_callback);
    add_timer(&tw, 5000, my_callback);

    clock_t start = clock();
    while (((clock() - start) * 1000 / CLOCKS_PER_SEC) < 6000) {
        tick(&tw);
        struct timespec ts = {0, RESOLUTION_MS * 1000000};
        nanosleep(&ts, NULL);
    }
    return 0;
}
```

Explicação rápida:

- O **Hierarchical Timing Wheel** organiza temporizadores em múltiplos níveis, onde cada nível cobre intervalos de tempo maiores.
- Cada nível é uma “roda” com um número fixo de slots (listas ligadas), onde temporizadores são colocados baseados no tempo que irão expirar.
- O ponteiro de cada nível avança em ticks regulares (exemplo: a cada 100 ms).
- Quando o ponteiro de um nível completa uma volta, temporizadores que não expiraram são promovidos para níveis inferiores com maior resolução.

Conclusão

O algoritmo Hierarchical Timing Wheel representa um avanço significativo no gerenciamento eficiente de temporizadores, resolvendo um problema clássico em ciência da computação com uma solução inspirada no tempo real e análoga a um relógio analógico.

Apesar de algumas limitações de granularidade e parametrização, seu uso é recomendado sempre que houver necessidade de escalabilidade e desempenho em sistemas com milhares ou milhões de temporizadores.

Referências

- Varghese, G., & Lauck, R. (1996). *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*. IEEE/ACM Transactions on Networking, 4(3), 348–356.
- Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (4th ed.). Pearson. (Seção sobre gerenciamento de temporizadores e escalonamento)
- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley. (Capítulo sobre temporizadores e gerenciamento de recursos)