Lua Tasks
Relatório INF2102

Thiago Duarte Naves

1 Introdução

Nesse projeto implementamos um módulo Lua com o objetivo de facilitar o desenvolvimento de aplicações concorrentes e reativas. Para tal, são fornecidas interfaces que ajudem a implementar blocos concorrentes e paralelos, eventos, futuros, callbacks e cancelamento de tarefas. Como Lua executa em apenas uma thread, não há execução de código Lua em paralelo. Paralelismo aqui se refere a trechos de código que possam coordenar chamadas externas assíncronas ou a código que possa ser executado como reação a estímulos externos.

Esse projeto foi inspirado pela linguagem de programação Céu [3] e sua forma de explicitar partes paralelas da aplicação utilizando a estrutura da linguagem. Os "blocos paralelos" neste módulo, juntamente com a interface de eventos, permitem semelhante expressividade em Lua.

Problemas diferentes são melhor resolvidos com ferramentas diferentes. Assim, queremos que todas as interfaces disponíveis nesse módulo possam ser usadas ao mesmo tempo pela aplicação. É essencial, portanto, que a interação entre estas seja não só suportada, mas natural. As interfaces de eventos e tarefas formam a base de todas as funcionalidades fornecidas sobre as quais as demais APIs foram construídas. Outras, como os blocos paralelos, são de mais alto nível e abstraem detalhes de cenários comuns.

2 Especificação

2.1 Integração

O módulo desenvolvido deve ser auto-contido e fornecer interfaces para integração com um escalonador externo. Deve ser possível gerar e consumir eventos de fora da aplicação, independentemente do uso de tarefas. Também deve haver uma interface para atualização da hora corrente. Dessa forma é possível utilizar o módulo como adaptador entre um escalonador e a aplicação do usuário, potencialmente unificando APIs diferentes.

Como exemplo de tal integração, o framework de jogos Löve 2D [1] fornece um escalonador com uma interface baseada em callbacks. Suas callbacks notificam a aplicação de eventos como mouse e teclado e do tempo decorrido. Juntamente com funções de desenho, essas interfaces permitem explorar o uso do módulo como descrito acima na construção de aplicações interativas.

2.2 Classes de base

- **Tarefa**: Estrutura para permitir a execução de código de forma concorrente. Permite parar, retomar e encerrar a execução de um trecho de código em função de eventos. Podem ser aninhadas de forma hierárquica, de forma que o término da tarefa mais externa encerra a execução das tarefas mais internas, denominadas sub-tarefas.

- **Evento**: Objeto que permite o envio de mensagens para um conjunto de tarefas ou callbacks. Pode ser utilizado em conjunto com as tarefas ou de forma independente. Permite a troca de mensagens entre tarefas bem como entre tarefas e outras partes da aplicação. Em particular, pode ser utilizado para notificar tarefas de eventos externos, como interação com o usuário e passagem de tempo.

- **Futuro**: Objeto que permite aguardar uma execução assíncrona ou um evento, bem como abortar essa execução. Pode ser utilizado juntamente com tarefas ou de forma independente. Quando utilizado por uma tarefa, permite que essa aguarde o término de uma execução sem bloquear imediatamente. A qualquer momento a tarefa pode utilizar o futuro para bloquear até o fim da

execução (ou emissão do evento). Esse objeto também armazena o valor retornado pelo evento associado.

- **Timer**: Objeto que permite executar ações em função da passagem do tempo. Fornece uma interface de callback para executar uma função após uma determinada quantidade de tempo ou em intervalos regulares.

2.3 Interface de blocos concorrentes

- **par_and**: Função que permite iniciar um conjunto de tarefas concorrentemente e aguardar até que todas terminem. Retorna uma nova tarefa que inicia as demais como sub-tarefas e bloqueia as aguardando.
- **par_or**: Função que permite iniciar um conjunto de tarefas concorrentemente e aguardar até que qualquer uma delas termine, encerrando a execução das demais. Retorna uma nova tarefa que inicia as demais como sub-tarefas e bloqueia as aguardando.
- **emit**: Função que executa um evento global, opcionalmente enviando dados através do mesmo.
- **await**: Função que bloqueia a execução de uma tarefa até que o evento global correspondente seja executado através da função **emit**. Retorna os dados enviados através do evento.
- **await_ms**: Função que bloqueia a execução de uma tarefa durante um determinado número de milissegundos.

2.4 Interface de callbacks

- **emit**: Função que executa um evento global, opcionalmente enviando dados através do mesmo.
- **listen**: Função que registra uma função como `listener` para ser executada quando um evento global for executado através da função **emit**. Os dados enviados para o evento serão passados como parâmetro da função.
- **stop_listening**: Função que remove um `listener` de um evento global.
- **in_ms**: Função que instancia um timer que executará uma função daqui a um determinado número de milissegundos. Retorna o objeto timer.
- **every_ms**: Função que instancia um timer que executará uma função regularmente com o intervalo determinado (em milissegundos) a partir de agora. Retorna o objeto timer.

2.5 Outras funções

- **update_time**: Função que informa ao módulo a quantidade de tempo decorrido desde a sua última execução (ou do início do programa, na primeira vez).
- **now_ms**: Função que retorna o `timestamp` atual. Essa contagem é dependente das execuções de **update_time** e não tem qualquer relação com o horário do sistema operacional.

3 Implementação

3.1 Mecanismos utilizados

A tabela (`Lua table`), um array associativo, é a principal estrutura de dados em Lua. Uma meta-tabla é uma tabela na qual algumas chaves especiais permitem alterar o comportamento de outra tabela associada além de implementar sobrecarga de operadores. Por exemplo, a chave `__call` da meta-tabela de uma tabela **A** permite definir uma função que será executada quando tentarmos executar **A()** (equivalente ao método `operator()()` em uma classe de C++).

Afim de prover uma API orientada a objetos, utilizamos uma combinação de tabelas e meta-tabelas para representar objetos e classes. A chave `__index` em uma meta-tabela permite retornar valores para chaves que não existem na tabela associada. Para instanciar objetos, o construtor cria uma nova tabela, inserindo nela os membros do objeto e associando uma meta-tabla com `__index` apontando para a tabela

com os membros da classe. Assim, os métodos podem ser definidos uma única vez para a classe e acessados a partir de qualquer objeto dessa classe. Para que possamos acessar os membros do objeto nesses métodos, eles recebem como o primeiro parâmetro uma referência para o objeto. O operador ":" automatiza essa chamada. Assim, `objeto:foo(1, 2)` equivale a `classe.foo(objeto, 1, 2)`.

Muitas das APIs foram implementadas sobre as demais para possibilitar a criação de interfaces de mais alto nível ou simplesmente alternativas sem retrabalho. As classes de evento e tarefa formam a base das funcionalidades deste módulo, fornecendo controle de concorrência e troca de mensagens. A figura 1 ilustra a relação entre as interfaces.

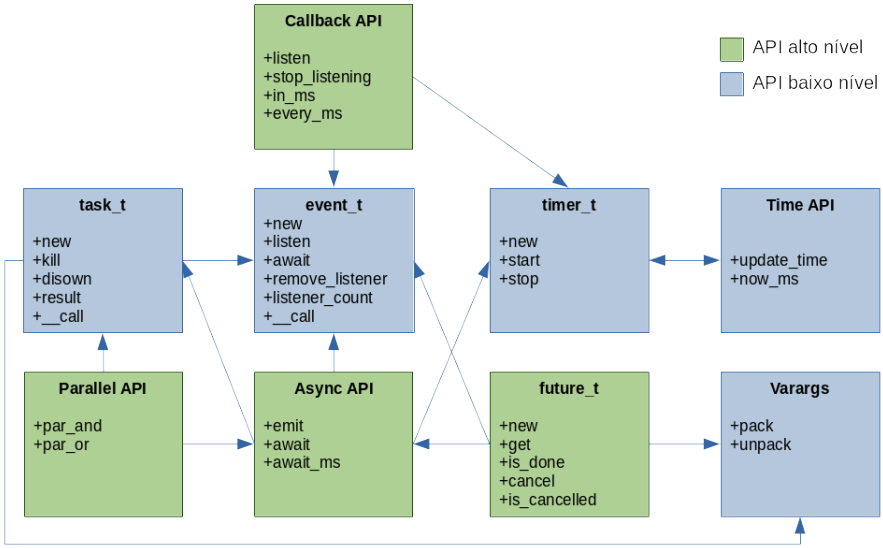

Figura 1: Dependências das APIs

3.2 API

- **event_t**:
  - Classe que implementa um objeto que quando executado (através do meta-método `__call`) executa todas as funções em sua lista de listeners. Ao inserir um listener pode-se escolher se este será executado sempre que esse evento executar ou apenas na execução seguinte à sua inserção. A inserção / remoção de listeners durante a execução do evento não tem efeito até o término da execução de todos os listeners. Isso impede que um listener inserido a partir de outro seja executado imediatamente, causando comportamentos inesperados. Apesar disso, `listener_count` reflete a mudança imediatamente.
- **event_t.listen(callback)**:
  - Adiciona um listener ao evento. Esse listener será executado toda vez que o evento for executado.
  - Se **callback** já for um listener do evento, troca seu "modo"para que seja executada todas as vezes que o evento executar.
  - Se esse método for executado a partir de um listener desse evento, insere a callback em uma lista de listeners (`new_listeners`) que serão considerados apenas a partir da próxima execução do evento. Caso contrário insere diretamente na lista de listeners do evento (`listeners`).
  - Se **callback** for um novo listener, incrementa `_listener_count`.
- **event_t.await(callback)**:
  - Adiciona um listener ao evento. Esse listener será executado apenas na próxima vez que o evento for executado.
  - Se **callback** já for um listener do evento, troca seu "modo"para que seja executada apenas na próxima vez que o evento executar.

  - Se esse método for executado a partir de um listener desse evento, insere a callback em uma lista de listeners (`new_listeners`) que serão considerados apenas a partir da próxima execução do evento. Caso contrário insere diretamente na lista de listeners do evento (`listeners`).
  - Se **callback** for um novo listener, incrementa `_listener_count`.

- **event_t.remove_listener(callback)**:
  - Remove a **callback** da lista de listeners do evento. Se a **callback** não for um listener desse evento, não faz nada. Se esse método for executado a partir de um listener desse evento, a remoção só será efetivada ao final da execução de todos os listeners.
  - Atualiza `_listener_count`, se necessário.
  - Se o evento está sendo executado, marca a **callback** para remoção.
  - Caso contrário, a remove imediatamente.

- **event_t.listener_count()**:
  - Retorna o número de listeners do evento (`_listener_count`).

- **event_t.__call(...)**:
  - Executa os listeners do evento, passando como parâmetro o objeto evento e os parâmetros passados para esse método.
  - Para cada listener:
    * Se for do tipo `"once"` (isto é, adicionado via `event_t.await`), remove-o da lista de listeners, decrementa `_listener_count` e o executa.
    * Se for do tipo `"repeat"` (isto é, adicionado via `event_t.listen`), apenas o executa.
  - Itera sobre a lista de listeners adicionados, modificados ou removidos durante a execução do evento e atualiza a tabela de listeners.

- **task_t**:
  - Classe que define uma tarefa: Abstração implementada sobre corrotinas que permite suspender e retomar uma execução em função de reações a eventos, bem como criar uma hierarquia de tarefas. Essa hierarquia permite encerrar sub-tarefas junto com a tarefa que as iniciou.

- **task_t.state**:
  - Indica o estado atual da tarefa:
    * `"ready"`: Ainda não iniciou a execução.
    * `"alive"`: Executando ou bloqueada.
    * `"dead"`: Finalizada.

- **task_t.new(f, name)**:
  - Construtor. Define a função que será executada pela tarefa (**f**) e o nome da tarefa (**name**).
  - **name** é usado para identificar a tarefa em mensagens de erro e em sua representação string (via `__to_string()`).

- **task_t.__tostring()**:
  - Meta-método que retorna uma string representando a tarefa contendo seu nome (`name`) e estado (`state`). Também usado pelas funções `tostring` e `print` da API padrão de Lua.

- **task_t.disown()**:
  - Dissocia a tarefa da tarefa pai. O término da tarefa pai não mais irá terminar esta.
  - Remove `suicide_cb` dos listeners da tarefa pai.
  - Deleta esse listener.
  - Remove a referencia para o pai (`parent`).

- **task_t.result()**:
  - Retorna os valores retornados pela função da tarefa (`f`).
  - Se a tarefa ainda não terminou, não retorna nada.
  - Caso contrário, utiliza a função `unpack` para desempacotar os múltiplos valores retornados que foram armazenados no objeto da tarefa e os retorna.
- **task_t._resume()**:
  - Método privado que retoma a execução da tarefa.
  - Guarda a referência da tarefa que está executando atualmente (`scheduler.current`).
  - Retoma a tarefa (`coroutine.resume`).
  - Em caso de erro (`coroutine.resume` retornou `false`), gera uma mensagem de erro a partir do erro ocorrido, da pilha de execução da tarefa e da pilha de tarefas que estão executando.
  - Propaga o erro.
  - Após o controle retornar da corrotina da tarefa (independente de erro), restaura a referência da tarefa que estava executando (`scheduler.current`).
  - Retorna um boolean indicando se a execução teve sucesso e a mensagem de erro
- **task_t.__call()**:
  - Meta-método que inicia a execução de uma tarefa. Bloqueia a tarefa que o executar até que a sub-tarefa termine. Retorna os valores retornados pela função do usuário (`f`). Recebe 2 parâmetros opcionais, `no_await` e `independent`. O primeiro faz com que essa chamada não bloqueie a execução da tarefa pai. O segundo faz com que a tarefa seja iniciada como uma tarefa independente (sem pai), não uma sub-tarefa. Nesse caso, o encerramento da tarefa que executou essa chamada não termina a execução da tarefa que foi iniciada.
  - No início desse método, a variável global `scheduler.current` indica a tarefa que está atualmente executando (se existir). Essa tarefa será considerada pai da tarefa que estamos iniciando.
  - Antes de executar a função do usuário (`f`), registramos um listener no evento `done` da tarefa pai. Esse mecanismo permite que as sub-tarefas sejam encerradas automaticamente com o fim da tarefa pai. Em seguida, trocamos `scheduler.current` para a sub-tarefa e iniciamos a corrotina (`_resume`).
  - Quando a execução retornar da corrotina, ou seja, quando esta terminar ou executar `await` / `await_ms`, inserimos um listener em seu evento `done` que emite um evento único. Em seguida, bloqueamos a tarefa pai até que esse evento seja emitido. Dessa forma, a execução da tarefa pai espera até que a sub-tarefa termine. Esse procedimento não é feito caso a sub-tarefa termine sem bloquear ou o parâmetro `no_await` seja `true`.
  - A função `f` é executada dentro de uma função lambda que captura seu retorno e ao final executa o método `kill` da sub-tarefa.
- **task_t.kill()**:
  - Método responsável por encerrar a execução de uma tarefa e executar o seu evento `done`.
  - Ao executar o evento `done`, todas as sub-tarefas são encerradas recursivamente. Caso esta seja a tarefa executando atualmente, executamos um `yield` para devolver o controle à tarefa que nos acordou.
- **par_and(a, b, c, ...)**:
  - Instancia uma tarefa que executa um grupo de tarefas concorrentemente e termina apenas quando todas terminam.
  - Um contador indica quantas das sub-tarefas ainda estão ativas. Iniciamos a tarefa pai e esta inicia as demais passando o parâmetro `no_await = true` em `__call`. Dessa forma todas iniciam concorrentemente e, em seguida, a tarefa pai bloqueia executando `await(UUID)`. `UUID` é um evento único para essa chamada `par_and`. Um listener registrado no evento `done` das sub-tarefas decrementa o contador e emite o evento `UUID` quando atinge 0, encerrando a tarefa pai.

  - Essa função aceita como parâmetro tarefas e funções. Para cada função é criada uma tarefa para que possa executar concorrentemente.

- **par_or(a, b, c, ...)**:
  - Instancia uma tarefa que executa um grupo de tarefas concorrentemente e termina quando qualquer uma delas terminar, retornando os mesmos valores retornados por esta.
  - Ao iniciar, a tarefa pai inicia as demais passando o parâmetro `no_await = true` em `__call`, de forma que todas iniciam em paralelo. Em seguida, bloqueia executando `await(UUID)`. `UUID` é um evento único para essa chamada `par_or`. Um listener registrado no evento `done` das sub-tarefas emite o evento `UUID`, causando o término da tarefa pai. Como o término de uma tarefa encerra a execução das suas sub-tarefas, as demais são terminadas automaticamente.
  - Essa função aceita como parâmetro tarefas e funções. Para cada função é criada uma tarefa para que executar concorrentemente.

- **await(event_id)**:
  - Cria (se necessário) um objeto `event_t` na tabela de eventos globais `_scheduler.waiting` com a chave `event_id`.
  - Adiciona a esse evento um listener que retomará a corrotina da tarefa atual (que executou o `await`).
  - Adiciona um listener ao evento `done` da tarefa atual para remover o listener do evento (evita vazamento de memória).
  - Faz `yield` da corrotina e retorna o retorno de `coroutine.yield`.

- **emit(event_id, [params ...])**:
  - Obtém o objeto `event_t` correspondente a `event_id` na tabela de eventos globais `_scheduler.waiting`.
  - Se existir executa o evento passando `[params ...]`.
  - Caso não hajam listeners registrados no evento, esse é destruído.

- **listen(event_id, callback, once)**:
  - Adiciona **callback** aos listeners do evento global identificado por **event_id**.
  - Se o evento não existir na tabela de eventos globais `_scheduler.waiting`, este será instanciado.
  - Se **once == true**, executa `event:await`, caso contrário `event:listen`.

- **stop_listening(event_id, callback)**:
  - Remove a **callback** dos listeners do evento global identificado por **event_id**.
  - Se o evento não existir ou **callback** não for um listener desse evento, não faz nada.

- **future_t**:
  - Objetos dessa classe representam processos que estejam ocorrendo em paralelo ou concorrentemente. Permitem que uma tarefa consulte o estado, aborte e obtenha o resultado do processo. Antes do término de tal processo, se uma tarefa tentar obter seu resultado (utilizando o método `get`), esta será bloqueada até que este esteja disponível.

    Internamente, um futuro aguarda um evento, inserindo um listener em sua lista que ira armazenar os dados (parâmetros) passados para o evento no objeto `future_t`.

- **future_t.new(event_id, [cancel_function])**:
  - Construtor. Associa o evento indicado ao futuro. Opcionalmente registra uma função de cancelamento para que o processo possa ser abortado se o futuro for cancelado.

- **future_t.get()**:
  - Retorna os valores passados para o evento correspondente. Caso o evento ainda não tenha sido emitido, bloqueia a tarefa aguardando o evento.

  - Verifica se o evento já foi emitido e se sim, retorna os dados já armazenados. Se não, utiliza `await` para bloquear em um evento interno único para este objeto. Esse evento será emitido pelo listener do evento associado a esse futuro.
  - Se o futuro foi cancelado, não retorna nada.

- **future_t.cancel()**:
  - Aborta o processo associado ao futuro e desbloqueia qualquer tarefa que esteja bloqueada no método `get` desse futuro.
  - Muda o estado do futuro (`state`) para `"cancelled"`.
  - Remove o listener do evento associado ao futuro, executa a função de cancelamento e emite o evento interno do futuro.

- **future_t.is_cancelled()**:
  - Verifica se o futuro foi cancelado.
  - Retorna `true` se o estado (`state`) for `"cancelled"`.

- **future_t.is_done()**:
  - Verifica se o futuro terminou (evento foi emitido).
  - Retorna `true` se o estado (`state`) for `"done"`.

- **timer_t**:
  - Classe responsável por criar temporizadores que permitem executar callbacks em função da passagem do tempo.

- **timer_t.new(interval, callback, cyclic)**:
  - Construtor. Define o intervalo de tempo entre o início da temporização e o momento em que a callback deve ser executada.
  - O parâmetro `cyclic` define se o temporizador deve executar a callback uma vez e parar (`false`) ou se deve executa-la periodicamente (`true`).

- **timer_t.start()**:
  - Inicia a contagem do intervalo de tempo definido no construtor.
  - Somamos `interval` com o timestamp atual e inserimos o temporizador na heap `_scheduler.waiting_time`.

- **timer_t.stop()**
  - Para a contagem do timer.
  - Remove o timer da heap (`_scheduler.waiting_time`), se necessário.

- **timer_t._execute()**:
  - Método privado que executa a callback do timer e o re-escalona, se necessário.

- **update_time(dt)**:
  - Função que informa a passagem do tempo ao módulo. Verifica na heap `_scheduler.waiting_time` se o novo timestamp é maior ou igual àquele no topo. Em caso positivo, removemos esse timer da heap, executamos sua callback e verificamos novamente a heap. Repetimos até que o próximo timestamp seja estritamente maior que o atual ou a heap esteja vazia.

- **in_ms(interval, callback)**:
  - Cria um objeto `timer_t` que ira executar a callback após o intervalo de tempo definido, iniciando a contagem imediatamente.
  - Retorna o timer criado para permitir o cancelamento de sua execução antes do fim do intervalo.

- **every_ms(interval, callback)**:

  - Cria um objeto `timer_t` que ira executar a callback periodicamente com o intervalo de tempo definido, iniciando a contagem imediatamente.
  - Retorna o timer criado para permitir o cancelamento de sua execução.
- **await_ms(interval)**:
  - Função que bloqueia a tarefa atual por um determinado número de milissegundos.
  - Cria um objeto `event_t` que irá retomar a corrotina da tarefa quando executado.
  - Utiliza `in_ms` para criar escalonar um timer que irá emitir o evento.
  - Bloqueia a tarefa utilizando o mesmo mecanismo do `await` para aguardar o evento criado.
- **now_ms()**:
  - Retorna o timestamp atual (`_scheduler.timestamp`).
- **pack(...)**
  - Agrupa um número variável de argumentos em uma tabela Lua, preservando valores nulos. Permite armazenar um número variável de parâmetros como um único valor.
  - Coloca o número de parâmetros como o primeiro valor na tabela (índice 0), seguido dos valores dos parâmetros (índices 1 à n). Armazenar o número de parâmetros é necessário para preservar nulos.
  - Retorna a tabela gerada, que deve ser extraída com a função `unpack()`.
- **unpack(t)**
  - Extrai e retorna os valores armazenados em uma tabela gerada pela função `pack()`.
- **_scheduler.timestamp**:
  - Variável que armazena o relógio que todas as funções de temporização seguem.
- **_scheduler.waiting_time**:
  - Heap ordenada por timestamp que permite obter de forma eficiente o próximo temporizador a ser tratado ao atualizar o timestamp atual.
- **_scheduler.current**:
  - Variável que armazena uma referência para a tarefa atualmente em execução.
- **_scheduler.waiting**:
  - Tabela que contém as instâncias de eventos globais.
  - Utiliza como chave o identificador do evento e como valor objeto `event_t` correspondente.

4 Testes

4.1 Testes automáticos

Testes unitários foram escritos ao longo do desenvolvimento do módulo a fim de exercitar todas as APIs e prevenir a reincidência de bugs. A maioria dos testes utiliza apenas APIs públicas e observa efeitos externos da execução. Contudo, em alguns casos mostrou-se necessário acessar membros privados do módulo ou de objetos para que fosse possível observar alguns efeitos, como vazamento de memória.

Não foi utilizada nenhuma ferramenta pronta para executar os testes automáticos. Em vez disso, um pequeno script (`tests.lua`) itera sobre uma lista de funções de teste, capturando erros e reportando no terminal o resultado de cada teste e ao final se todos tiveram sucesso ou quantos falharam. Essa solução se mostrou suficiente e permite que os testes sejam autocontidos, sem necessitar da instalação ou inclusão de uma ferramenta externa. Outra vantagem é possibilitar o teste do módulo em diferentes interpretadores Lua, seja em versões diferentes do interpretador oficial, seja em versões de terceiros, como o LuaJIT [2].

O módulo desenvolvido foi testado em duas versões do interpretador Lua: 5.1 e 5.3. Os mesmo programa de testes foi executado com sucesso em ambas as versões.

4.2 Exemplos

Os exemplos de uso do módulo utilizam o framework Löve 2D como interface para interação com o usuário. Essa é uma plataforma popular que provê várias funcionalidades úteis para exercitar o uso das APIs desenvolvidas: Funções gráficas permitem uma interação mais rica com o usuário. Animações e mudanças gráficas ilustram com clareza mudanças de estados, especialmente ao lidar com múltiplos ao mesmo tempo. Callbacks de eventos de mouse e teclado permitem a fácil interação com o mundo externo. Um loop interno assíncrono informa a passagem do tempo.

Os exemplos também serviram de teste manual durante o desenvolvimento e como uma forma de explorar as APIs em um contexto mais geral. Testes amplos como estes ajudaram a explorar interações mais complexas entre as APIs, inclusive revelando bugs que foram solucionados e incluídos nos testes automáticos.

A listagem 1 apresenta um exemplo no qual criamos uma tarefa que pisca um LED (representado por um círculo cheio ou vazio) uma vez por segundo até que a barra de espaço seja pressionada.

```
1  local tasks = require("tasks")
2  local main_task
3  local led1 = true
4
5  function love.load()
6      -- main
7      function blink()
8          while true do
9              tasks.await_ms(500)
10             led1 = not led1
11         end
12     end
13
14     main_task = tasks.par_or(blink, function() tasks.await("space") end)
15     main_task()
16 end
17
18 function love.update(dt)
19     tasks.update_time(dt * 1000)
20 end
21
22 function love.keypressed(key, scancode, isrepeat)
23     if isrepeat then return end
24     tasks.emit(key)
25 end
26
27 function love.draw()
28     love.graphics.setColor(1, 1, 1) -- White
29     --                                              x    y    r   segments
30     love.graphics.circle(led1 and "fill" or "line", 100, 100, 20, 100)
31 end
```

Listagem 1: Exemplo pisca LED

4.3 Casos de uso

O módulo, juntamente com o Löve 2D, foi utilizado pelos alunos da disciplina Sistemas Reativos para implementar o clássico jogo *Genius* em aula. Nenhum dos grupos relatou problemas nem grandes dificuldades em seu uso. Além disso, alguns alunos relataram que as funções fornecidas são uma forma mais fácil de programar do que utilizando apenas as APIs do Löve e manifestaram interesse em utiliza-las no trabalho final da disciplina.

5 Manual do usuário

5.1 Blocos concorrentes

- **Tarefas (classe task_t)**: Extensão de corrotina que permite que funções sejam executadas em paralelo e canceladas.
  - Uma referência para esse objeto deve ser mantida até que sua execução termine.
  - Tarefas iniciadas por outra tarefa são consideradas sub-tarefas.
  - Quando uma tarefa termina (ou é cancelada), todas as suas sub-tarefas são canceladas.
  - Uma tarefa não pode ser reiniciada depois de terminar.
  - **t = task_t:new(f, [name])**: Construtor.
    - Define **f** como a função que será executada pela corrotina.
    - **name** é usado para debug, identifica a tarefa em mensagens de erro.
  - **t([no_await], [independent])**: Inicia a corrotina.
    - Por padrão, bloqueia até que a tarefa termine (apenas se executada dentro de outra tarefa).
    - **no_await**: Se true, não aguarda o término da nova tarefa, retornando quando esta chamar **await**. Não tem efeito se for executado de fora de uma tarefa.
    - **independent**: Se true, a tarefa será iniciada como uma tarefa independente, não como sub-tarefa. Dessa forma, o término da tarefa que fez a chamada não terminará essa tarefa.
    - Essa função retorna os valores retornados por **f** (apenas se **no_wait** $\neq$ **true** ou **f** retornar imediatamente).
  - **t:result()**: Retorna os valores retornados por **f**.
    - Não retorna nada se chamada antes de **f** retornar.
  - **t:disown()**: Torna essa tarefa independente da tarefa que a iniciou, deixando se ser uma sub-tarefa. Equivalente a usar **independent = true** no construtor. Só deve ser executado antes da tarefa iniciar ou se ela foi iniciada com **no_wait = true**.
  - **t.state**: Indica o estado atual da tarefa:
    - `"ready"`: Ainda não iniciou a execução.
    - `"alive"`: Executando ou bloqueada.
    - `"dead"`: Finalizada.
- **emit(id, [dados, ...])**: Emite o evento identificado por **id**.
  - Desbloqueia todas as tarefas que chamaram **await(id)**.
  - Executa as callbacks registradas com **listen(id, callback)**.
  - Se o evento correspondente não existir ou não tiver listeners, não faz nada.
  - Todos os parâmetros passados após **id** ([**dados, ...**]) serão propagados para as callbacks e **awaits**.
  - Note que não há garantia da ordem em que as callbacks serão executadas e tarefas desbloqueadas, podendo inclusive intercalar callbacks e desbloqueios.
  - Emitir um evento de dentro de um de seus listeners não é suportado.
  - Executar **emit** do mesmo evento usado no último **await** em uma tarefa não é suportado. Isso é, não é possível executar **emit** do evento ao qual estamos reagindo.

- **await(id)**: Bloqueia a tarefa atual até que o evento identificado por **id** seja emitido.
  - Retorna os [**dados, ...**] passados por parâmetro na emissão do evento.
- **par_or(a, b, [c, [...]])**: Inicia as tarefas passadas por parâmetro. Se qualquer delas terminar, todas as demais serão encerradas.
  - **a, b, c, ...** devem ser funções ou objetos do tipo **task_t**.
  - Para permitir a execução concorrente, tarefas serão criadas automaticamente para as funções.
  - O retorno da execução (meta-método **__call**) da tarefa retornada é o mesmo da sub-tarefa que terminar primeiro.
- **par_and(a, b, [c, [...]])**: Inicia as tarefas passadas por parâmetro e aguarda até que todas terminem.
  - **a, b, c, ...** devem ser funções ou objetos do tipo **task_t**.
  - Para permitir a execução concorrente, tarefas serão criadas automaticamente para as funções.

5.2 Callbacks

- **listen(id, callback, [once])**: Insere a **callback** na lista de listeners do evento identificado por **id**.
  - Se o evento não existir, ele é criado.
  - **once**: Se true, a **callback** só será executada na próxima execução (**emit**) do evento. Caso contrário, executará toda vez que o evento executar.
  - Se **callback** já estiver na lista de callbacks desse evento, atualiza a propriedade **once** para o desta chamada.
  - Executar **emit** de um evento a partir de uma callback do mesmo evento não é suportado.
- **stop_listening(id, callback)**: Remove a **callback** da lista de listeners do evento identificado por **id**.
  - Se o evento não existir ou **callback** não for um listener do evento, não faz nada.

5.3 Futuros

- **Futuro (classe future_t)**: Aguarda a execução de um evento sem bloquear.
  - **f = future_t:new(id, [cancel_cb])**: Construtor. Instancia um objeto que espera o evento identificado por id.
  - **f:get()**: Retorna os dados do evento.
    - Se o evento ainda não foi emitido, bloqueia a tarefa até que ele seja emitido.
  - **f:cancel()**: Para de aguardar o evento e executa **cancel_cb**.
  - **f:is_done()**: Permite testar se o evento já foi emitido.
  - **f:is_canceled()**: Retorna true se o futuro foi cancelado.

5.4 Timers

- **Timer (classe timer_t)**: Controla a execução de uma callback em função do tempo.
  - **t = timer_t:new(interval, callback, [cyclic])**: Construtor.
    - **interval**: Tempo a ser contado em milisegundos.
    - **callback**: Função a executar.
    - **cyclic**: Se true, a **callback** será executada a cada **interval** ms, caso contrário, será executada apenas uma vez após **t:start()**.
  - **t:start()**: Inicia a contagem do tempo para a execução da **callback**.
    - Se já estiver contando, não faz nada.

        * Reinicia um timer parado.
    - **t:stop()**: Para o timer.
        * Se já estiver parado, não faz nada.

- **now_ms()**: Retorna o timestamp atual em milisegundos.
    - Esse timestamp é interno a esse módulo e não tem relação com o horário do sistema operacional.
- **in_ms(ms, callback)**: Executa a **callback** daqui a ms milisegundos.
    - Retorna o objeto **timer_t** que controla a contagem.
- **every_ms(ms, callback)**: Executa a **callback** a cada ms milissegundos a partir de agora.
    - Retorna o objeto **timer_t** que controla a contagem.
- **await_ms(ms)**: Bloqueia a tarefa por **ms** milisegundos.

6 Considerações finais

O projeto apresentou bons resultados e demonstra a viabilidade do uso simultâneo de diferentes estilos de interface em uma mesma aplicação. Há espaço para que diversas outras interfaces possam ser adicionas às que foram implementadas nesse módulo permitindo maior flexibilidade nas aplicações. Por exemplo, estruturas de dados ou variáveis observáveis e visões sobre dados armazenados nelas (como em um banco de dados). Além disso, novos métodos podem ser adicionados às classes desenvolvidas, como um futuro coordenando uma tarefa, o reinício de tarefas e espera de múltiplos eventos simultaneamente.

Em outra frente, a implementação pode ser aprimorada. Em particular, diminuindo a quantidade de alocações de memória feitas em estruturas de controle internas. Devido ao uso do coletor de lixo, menos alocações também resultariam em menos tempo de CPU gasto na coleta de lixo. Erros dentro de tarefas geram mensagens desnecessariamente longas, frequentemente contendo a mesma informação múltiplas vezes.

Pretendo continuar o desenvolvimento deste módulo na dissertação de mestrado e abordar os problemas descritos acima, bem como expandir as funcionalidades existentes.

Referências

[1] *Löve 2D*. URL: `https://love2d.org`. (acessado: 05.06.2020).

[2] *LuaJIT*. URL: `https://luajit.org`. (acessado: 05.06.2020).

[3] Francisco Sant'Anna et al. "The design and implementation of the synchronous language Céu". Em: (2017). DOI: `http://dx.doi.org/10.1145.3035544`.

A Log de testes - Lua 5.3

```
$ lua5.3 tests.lua

event_await (1/69): OK
event_listen (2/69): OK
event_remove_listener (3/69): OK
event_listener_count (4/69): OK
event_add_listener_in_callback (5/69): OK
event_add_await_in_callback (6/69): OK
event_remove_listener_in_callback (7/69): OK
event_remove_await_in_callback (8/69): OK
event_add_and_remove_listener_in_callback (9/69): OK
tasks_start_with_state_ready (10/69): OK
tasks_set_current_task (11/69): OK
emit_without_await_does_nothing (12/69): OK
emit_unblocks_await (13/69): OK
await_returns_emit_params (14/69): OK
task_kill_removes_await_listener (15/69): OK
task_kill_unblocks_parent (16/69): OK
task_kill_ends_par_or (17/69): OK
task_kill_counts_towards_par_and (18/69): OK
task_suicide_unblocks_parent (19/69): OK
task_suicide_ends_par_or (20/69): OK
task_suicide_counts_towards_par_and (21/69): OK
inner_task_blocks_outer_task (22/69): OK
outer_task_kills_inner_task (23/69): OK
inner_task_removes_done_listener_from_parent_on_kill (24/69): OK
task_no_wait_execution (25/69): OK
par_or_finishes_with_either (26/69): OK
par_and_finishes_with_both (27/69): OK
nested_par_or (28/69): OK
nested_par_and (29/69): OK
par_and_in_par_or (30/69): OK
par_and_in_par_or (31/69): OK
three_nested_ors (32/69): OK
three_nested_ands (33/69): OK
par_or_three_tasks (34/69): OK
par_and_three_tasks (35/69): OK
par_or_function_parameter (36/69): OK
par_and_function_parameter (37/69): OK
independent_subtask (38/69): OK
disown_subtask (39/69): OK
listen_repeat (40/69): OK
listen_once (41/69): OK
future_get_blocks (42/69): OK
future_get_doesntblock (43/69): OK
future_get_multiple_returns (44/69): OK
future_done (45/69): OK
future_get_cancelled (46/69): OK
future_get_cancelled_ignores_event (47/69): OK
future_cancel_done (48/69): OK
future_cancel_cb_executes_on_cancel (49/69): OK
future_cancel_cb_doesnt_execute_on_double_cancel (50/69): OK
future_cancel_cb_doesnt_execute_after_done (51/69): OK
task_return_value (52/69): OK
task_result (53/69): OK
par_or_return_value (54/69): OK
par_or_handles_subtask_not_blocking (55/69): OK
```

```
par_and_handles_all_subtasks_not_blocking (56/69): OK
par_and_handles_subtask_not_blocking (57/69): OK
trigger_once_timer_executes_only_once (58/69): OK
cyclic_timer_executes_twice (59/69): OK
trigger_once_timer_executes_after_2_updates (60/69): OK
cyclic_timer_executes_after_2_updates (61/69): OK
trigger_once_timer_executes_when_late (62/69): OK
cyclic_timer_executes_when_late (63/69): OK
stop_trigger_once_timer (64/69): OK
stop_cyclic_timer (65/69): OK
in_ms_triggers_only_once (66/69): OK
every_ms_triggers_periodically (67/69): OK
await_ms_blocks_task (68/69): OK
await_ms_unblock_multiple_tasks_at_once (69/69): OK
-----------------------------------------
All tests were successfull
```

B Log de testes - Lua 5.1

```
$ lua5.1 tests.lua

event_await (1/69): OK
event_listen (2/69): OK
event_remove_listener (3/69): OK
event_listener_count (4/69): OK
event_add_listener_in_callback (5/69): OK
event_add_await_in_callback (6/69): OK
event_remove_listener_in_callback (7/69): OK
event_remove_await_in_callback (8/69): OK
event_add_and_remove_listener_in_callback (9/69): OK
tasks_start_with_state_ready (10/69): OK
tasks_set_current_task (11/69): OK
emit_without_await_does_nothing (12/69): OK
emit_unblocks_await (13/69): OK
await_returns_emit_params (14/69): OK
task_kill_removes_await_listener (15/69): OK
task_kill_unblocks_parent (16/69): OK
task_kill_ends_par_or (17/69): OK
task_kill_counts_towards_par_and (18/69): OK
task_suicide_unblocks_parent (19/69): OK
task_suicide_ends_par_or (20/69): OK
task_suicide_counts_towards_par_and (21/69): OK
inner_task_blocks_outer_task (22/69): OK
outer_task_kills_inner_task (23/69): OK
inner_task_removes_done_listener_from_parent_on_kill (24/69): OK
task_no_wait_execution (25/69): OK
par_or_finishes_with_either (26/69): OK
par_and_finishes_with_both (27/69): OK
nested_par_or (28/69): OK
nested_par_and (29/69): OK
par_and_in_par_or (30/69): OK
par_and_in_par_or (31/69): OK
three_nested_ors (32/69): OK
three_nested_ands (33/69): OK
par_or_three_tasks (34/69): OK
par_and_three_tasks (35/69): OK
par_or_function_parameter (36/69): OK
par_and_function_parameter (37/69): OK
```

```
independent_subtask (38/69): OK
disown_subtask (39/69): OK
listen_repeat (40/69): OK
listen_once (41/69): OK
future_get_blocks (42/69): OK
future_get_doesntblock (43/69): OK
future_get_multiple_returns (44/69): OK
future_done (45/69): OK
future_get_cancelled (46/69): OK
future_get_cancelled_ignores_event (47/69): OK
future_cancel_done (48/69): OK
future_cancel_cb_executes_on_cancel (49/69): OK
future_cancel_cb_doesnt_execute_on_double_cancel (50/69): OK
future_cancel_cb_doesnt_execute_after_done (51/69): OK
task_return_value (52/69): OK
task_result (53/69): OK
par_or_return_value (54/69): OK
par_or_handles_subtask_not_blocking (55/69): OK
par_and_handles_all_subtasks_not_blocking (56/69): OK
par_and_handles_subtask_not_blocking (57/69): OK
trigger_once_timer_executes_only_once (58/69): OK
cyclic_timer_executes_twice (59/69): OK
trigger_once_timer_executes_after_2_updates (60/69): OK
cyclic_timer_executes_after_2_updates (61/69): OK
trigger_once_timer_executes_when_late (62/69): OK
cyclic_timer_executes_when_late (63/69): OK
stop_trigger_once_timer (64/69): OK
stop_cyclic_timer (65/69): OK
in_ms_triggers_only_once (66/69): OK
every_ms_triggers_periodically (67/69): OK
await_ms_blocks_task (68/69): OK
await_ms_unblock_multiple_tasks_at_once (69/69): OK
-------------------------------------------
All tests were successfull
```