5
Implementação

Neste capítulo apresentamos a implementação do modelo de eventos e multitarefa com as características descritas no capítulo 4. A implementação deve cobrir a utilização de threads bem como processos. Queremos disponibilizar as duas formas de multitarefa, deixando a cargo do desenvolver escolher a mais adequada para cada parte da aplicação.

Pela figura 4.5, vemos que um processo Lua, como nossa unidade básica de execução, pode interagir com outros processos, estejam eles no mesmo processo do sistema operacional ou não. Um dos objetivos do trabalho é explorar as características de cada ambiente, principalmente da memória compartilhada, na interação. Por exemplo, Johansson *et al.* (Johansson et al., 2002) discutem como estratégias de memória contribuíram para ganhos de desempenho na implementação da comunicação dos processos de Erlang. Então, as duas formas de interação possuem suas características, que queremos explorar na implementação.

Utilizamos o ALua como base para a construção do modelo proposto. Atualmente, o ALua permite a interligação de programas Lua, cada um executando em processos diferentes do sistema operacional. O ALua tem a vantagem de utilizar canais TCP na comunicação, o que permite a interação de processos em máquinas diferentes.

A nossa estratégia foi montar um modelo incremental, iniciando pela interação dos estados Lua dentro de um processo, e sobre isso, criar a interação dos estados entre processos, adaptando o modelo do ALua. A idéia é que dentro de um processo, possamos explorar o acesso direto à memória para criar essa interação, depois usar um mecanismo do sistema operacional para preencher a lacuna de comunicação entre processos.

5.1
Concorrência Intra-processo

Desenvolvemos uma biblioteca, `ccr` (*concurrency*), em C que é responsável por prover as operações básicas necessárias para a criação de estados Lua e a comunicação entre eles. A biblioteca, ao ser carregada pelo programa principal Lua, dispara um conjunto de threads, as quais ficarão responsáveis por executar

os processos Lua.

Pela nossa proposta, a comunicação se dará via troca de mensagens assíncronas (eventos) entre os processos Lua. Assim, definimos que cada processo terá a sua própria fila de mensagem, o que facilita a remoção das mensagens e reduz concorrência. Por exemplo, o mecanismo de *linda* provido por Lanes é compartilhado por todos os participantes. Mesmo que dois deles não interajam diretamente, eles podem disputar o mesmo recurso para receber ou enviar mensagens.

5.1.1 Criação e Identificação dos Estados Lua

Os novos processos Lua são criados pela função `ccr.spawn`, que recebe uma string contendo o programa Lua a ser executado. Um processo Lua corresponde a um estado Lua e uma estrutura de controle. Essa estrutura contém a fila de mensagens e também será usada para representar o processo Lua diante dos demais. Após a criação, a função carrega as bibliotecas padrão, a própria biblioteca `ccr` e o programa Lua. O processo Lua é então adicionado em uma fila interna de processos prontos para execução. As threads ociosas verificam essa fila a fim de recuperar um processo para execução. A figura 5.1 mostra uma simplificação da função `ccr.spawn` — `proc` é a estrutura de controle.

```
static int ccr_spawn(lua_State *L)
{
  process_t *proc;
  const char *code = lua_tostring(L, 1);

  proc = new process_t();      /* Representação interna */
  proc->L = luaL_newstate();   /* Novo estado Lua       */
  proc->queue = new queue_t(); /* Fila de mensagens     */

  /* Inicializa outros campos de 'proc' e     */
  /* e carrega as bibliotecas no novo estado. */

  /* Carrega o programa Lua */
  luaL_loadstring(proc->L, code);
  /* Insere na fila global de prontos para execução  */
  ready_queue.push(proc);

  /* Retorna para o chamador uma referência ao novo estado */
  create_ref(L, proc);
  return 1;
}
```

Figura 5.1: Pseudo-código da criação de um novo estado Lua.

A função `ccr.spawn` retorna a estrutura representando o novo processo Lua para o seu criador. Todos os processos têm acesso à sua própria identificação através da função `ccr.self`. A primitiva de envio de informação utiliza esta estrutura como

identificador de destinatário. Para tornar o acesso mais rápido à estrutura, ela é retornada para Lua como sendo do tipo *userdata*, o que a torna opaca aos programas Lua, mas dá acesso direto via C.

Essa estratégia tem a desvantagem de que não podemos passar esse identificador pelas mensagens trocadas entre os processos. Para contornar nossa limitação, disponibilizamos um serviço de registro na biblioteca onde podemos armazenar e consultar os identificadores utilizando uma string como chave. Essa chave pode ser enviada a outros participantes dentro do mesmo processo para que eles recuperem o identificador do estado. Acreditamos que o número de acessos ao registro será muito menor comparado com a troca de mensagens, uma vez que recuperada a identificação do estado o registro não é mais utilizado.

A estrutura de representação do processo é única e todos estão compartilhando a mesma referência. Utilizamos contagem de referências para controlar quando essa estrutura pode ser liberada. Acreditamos que essa contagem não terá um grande impacto, pois o contador é incrementado na criação e quando um outro estado obtém uma referência.

Um problema conhecido no uso de contagem de referência para coleta de lixo, que previne a coleta dos elementos, é a formação de ciclos. No entanto, quando um processo Lua termina a sua execução, ele remove as referências a outros processos e decrementa o seu próprio contador. Assim, a medida que os processos vão terminando, as referências a vão sendo removidas e em um certo momento o ciclo será desfeito. Além disso, se um processo enviar uma mensagem a outro processo, o qual já tenha encerrado sua execução, é reportado um erro e o emissor pode liberar a referência.

5.1.2 Comunicação em C

A estrutura utilizada para a troca de mensagens cumpre um papel importante em nosso modelo, tanto pelo desempenho no envio, mas também para a implementação do escalonamento com as threads (descrita mais adiante). Desenvolvemos em C uma versão de fila de mensagens seguindo um modelo clássico de uma lista encadeada protegida por um lock para garantir a consistência das operações de inserção e remoção (utilizamos as facilidades da biblioteca Threads POSIX (Butenhof, 1997)). Chamaremos essa implementação de *fila clássica*. Fizemos então a comparação com alguns mecanismos do sistema operacional que são geralmente utilizados em comunicação entre processos. Essa comparação serve para termos parâmetros de desempenho de nosso mecanismo de comunicação.

Como descrevemos anteriormente, Lua não permite o compartilhamento de seus elementos internos, então, as strings enviadas para outros estados Lua devem

ser copiadas. O teste da fila consiste em uma thread em C executando produtor e outra um consumidor, para simular uma situação de interação máxima entre duas partes. O produtor faz uma cópia de string padrão e a insere na fila do consumidor. Este por sua vez fica retirando as strings da fila constantemente e copiando-as para um buffer local, simulando de forma simples a inserção de uma string no estado Lua. De fato, Lua faz um *hash* da string no momento em que ela é inserida no estado. Nossa simulação leva menos tempo que a operação real, mas é consistente, pois todos realizam a operação do mesmo modo.

Entre os mecanismos de sistema operacional, testamos conexões TCP e sockets UNIX. Como no caso da fila, temos duas threads executando dois estados (produtor e consumidor) onde as strings são enviadas diretamente pelos canais do sistema operacional, delegando para este o controle de leitura e escrita, e lidas para um buffer interno do consumidor.

Os testes foram realizados em uma máquina com processador Pentium Core 2 Quad (quatro núcleos, cada um com 2,4GHz), executando Linux 64 bits, kernel 2.6.24. O produtor envia 100.000 mensagens ao consumidor, sendo cada programa executado 10 vezes para avaliação da média. A medida do tempo é feita desde o momento em que o produtor começa a enviar até o fim do recebimento de todas as mensagens. A tabela 5.1 mostra o tempo médio e o desvio padrão (σ) para tamanhos variados da mensagem.

Mensagem	**Clássico**		**TCP**		**UNIX**	
	Média	σ	Média	σ	Média	σ
64 bytes	224,74	66,80	63,78	11,03	187,77	22,94
128 bytes	214,85	76,87	65,12	3,43	193,06	26,45
256 bytes	163,91	51,64	83,03	8,89	170,18	25,49
512 bytes	198,16	58,63	121,57	18,79	206,28	25,11
1.024 bytes	284,40	87,04	163,94	9,21	155,48	15,17
2.048 bytes	273,30	56,76	251,22	46,91	200,51	1,24
4.096 bytes	447,07	69,63	625,71	28,01	374,07	3,48

Tabela 5.1: Tempo médio e os desvio padrão, em milissegundos, da troca de mensagens entre produtor e consumidor.

Verificamos que, no geral, a implementação clássica possuía um rendimento pior que TCP e socket UNIX. O desvio padrão também mostra que no caso da fila, temos uma variação maior no tempo. Essa variação se deve basicamente à concorrência pelo lock da fila e o escalonamento das threads. Decidimos então analisar estruturas *lock-free* para implementação de nossa fila. Utilizamos a biblioteca *Threading Building Blocks* (TBB) versão 2.1 (Intel, 2005), desenvolvida inicialmente pela Intel, que disponibiliza um conjunto de estruturas, dentre elas filas e tabelas hash. A tabela 5.2 mostra o tempo com o uso da fila lock-free e o gráfico na figura 5.2 ilustra esses valores.

Mensagem	Clássico		Lock-free		TCP		UNIX	
	Média	σ	Média	σ	Média	σ	Média	σ
64 bytes	224,74	66,80	57,09	31,28	63,78	11,03	187,77	22,94
128 bytes	214,85	76,87	81,66	37,52	65,12	3,43	193,06	26,45
256 bytes	163,91	51,64	78,99	34,44	83,03	8,89	170,18	25,49
512 bytes	198,16	58,63	115,72	72,10	121,57	18,79	206,28	25,11
1.024 bytes	284,40	87,04	158,34	64,01	163,94	9,21	155,48	15,17
2.048 bytes	273,30	56,76	204,19	47,88	251,22	46,91	200,51	1,24
4.096 bytes	447,07	69,63	415,55	14,27	625,71	28,01	374,07	3,48

Tabela 5.2: Tempos do produtor e consumidor com filas lock-free (milissegundos).

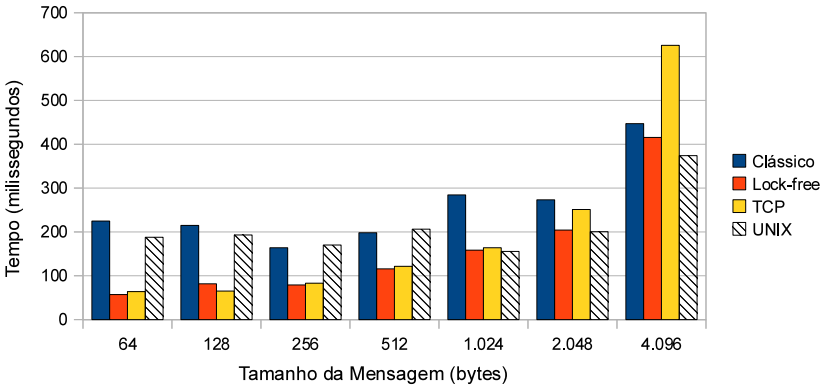

Figura 5.2: Gráfico com os tempos de troca de mensagens entre produtor e consumidor.

No entanto, os testes realizados de produtor e consumidor para as filas clássica e lock-free podem ser vistos como o pior cenário, onde temos uma competição constante de acesso. Pelo desvio padrão, podemos notar também que as filas apresentam uma variação maior nos tempos de execução, se comparado com TCP e socket UNIX.

Decidimos medir também o melhor caso para essas filas, quando apenas uma thread está acessando a fila — sem concorrência. Para isso, fizemos um laço de 100.000 iterações que consistia em inserir um mensagem e retirá-la logo em seguida. Os tempos médios, em milissegundos, são apresentados na tabela 5.3. Repetimos os valores de pior caso por motivo de comparação. Utilizamos *SC* (sem concorrência) para os novos valores medidos e *CC* (com concorrência) para o caso do acesso concorrente medido anteriormente.

Além do teste de comunicação ponto-a-ponto, decidimos exercitar também um modelo de comunicação *muitos-para-um*, a fim de verificar o comportamento dos mecanismos de comunicação. A tabela 5.4 mostra o tempo médio e o desvio padrão (σ), em milissegundos, para 10 produtores enviarem, cada um, 10.000 mensagens a um consumidor. Os valores mostram que a alocação de memória e cópia da string têm um impacto alto à medida que o tamanho da mensagem aumenta.

Mensagem	Clássico (SC)	Lock-free (SC)	Clássico (CC)	Lock-free (CC)
64 bytes	22,32	16,99	224,74	57,09
128 bytes	28,96	23,57	214,85	81,66
256 bytes	30,45	25,21	163,91	78,99
512 bytes	33,26	27,99	198,16	115,71
1.024 bytes	38,76	34,07	284,40	158,34
2.048 bytes	55,87	50,83	273,30	204,19
4.096 bytes	95,37	85,37	447,07	415,55

Tabela 5.3: Comparação dos tempos (milissegundos) de melhor e pior casos para o exemplo do produtor e consumidor.

Mensagem	Clássico		Lock-free	
	Média	σ	Média	σ
64 bytes	93,62	33,77	64,88	45,31
128 bytes	73,37	39,71	48,92	18,85
256 bytes	91,75	30,21	69,21	40,49
512 bytes	151,29	32,04	129,57	52,38
1.024 bytes	302,37	78,95	276,33	107,36
2.048 bytes	659,86	115,41	574,94	185,01
4.096 bytes	1.325,86	156,70	1.426,78	402,97

Tabela 5.4: Tempos, em milissegundos, para 10 produtores e um consumidor.

Diante desses testes, decidimos investigar mais o uso da fila clássica e principalmente da lock-free como base de comunicação. Mas, vale destacar que a fila lock-free também sofre com a variação de tempo, indicada pelo desvio padrão, causada pela concorrência — mesmo não que fila não tenha um lock, ainda assim há uma competição por inserir e remover os elementos. Além disso, os canais TCP e socket UNIX, por padrão, possuem um buffer interno muito mais limitado se comparado com a memória total do computador, o que leva ao bloqueio do emissor quando esse buffer está cheio. Em nosso computador de teste, a configuração padrão destina no máximo 4 Mbytes para o buffer de escrita do TCP. Com filas em memória temos mais liberdade com relação ao seu tamanho.

Esta também é uma oportunidade de investigar mais sobre o desenvolvimento com estruturas lock-free, que apresentou desempenho melhor que o modelo de lock tradicional, mas que exige uma maneira diferente de pensar sobre como estruturar a concorrência.

5.1.3 Comunicação em Lua e Escalonamento

A fila será a forma de comunicação entre os processos Lua dentro de um processo do sistema operacional e será utilizada como um indicador para suspender a execução dos processos Lua. Por exemplo, se o loop de eventos verificar a fila

para retirada de novos eventos, mas esta se encontra vazia, o processo Lua pode ser suspenso. Quando uma nova mensagem for postada na fila, o processo Lua é colocado na fila de prontos, para retomar seu processamento.

O esquema de bloqueio da fila clássica auxilia na implementação do comportamento descrito acima. As tarefas de suspensão e retomada requerem a realização de várias operações não atômicas, por isso, a abordagem de exclusão mútua da fila clássica nos permite fazer essas tarefas com mais facilidade. No caso de lock-free, não há limitador de concorrência e várias threads podem ter acesso simultâneo à fila, sendo difícil ter uma idéia momentânea do estado da fila.

Decidimos construir o mecanismo de comunicação usando os dois tipos de fila e então compará-los, pois tínhamos a impressão de que controlar o dinamismo da estrutura lock-free demandaria mais esforço computacional, que poderia levar a anulação dos ganhos apresentados até agora.

Como esperado, a implementação com a fila clássica foi mais simples, pois a exclusão mútua simplificava a tarefa de suspender e colocar novamente o estado Lua em execução.

Analisando essa primeira construção, vimos como tentar aproveitar a filas lock-free e locks de leitores e escritores (Andrews, 2000) para que pudéssemos montar uma política que explorasse a concorrência. O lock de escrita permitiria que vários escritores pudessem colocar mensagens na fila e a retirada de mensagens da fila não teria lock (visto que a estrutura da fila dá suporte à acesso múltiplo). Mas, quando fosse necessário suspender o estado Lua, o lock de leitura impedia qualquer acesso à fila, garantindo um estado consistente da mesma. Note que é um comportamento inverso do que geralmente é apresentado na literatura, onde locks de leitores e escritores oferecem suporte simultâneo de leitura e exclusivo de escrita.

A figura 5.3 mostra um pseudo-código de nossa implementação. Toda escrita passa por um lock, mas que permite vários escritores chegarem à fila de mensagens. Na linha 13, a leitura é realizada sem proteção, ficando a cargo da implementação da fila lock-free controlar o acesso concorrente. Somente se não houver mensagem, o lock de leitura é ativado (linha 16), bloqueando qualquer acesso por parte dos escritores e garantindo a consistência no procedimento `suspend`. Este é responsável por marcar o estado como bloqueado e liberar o lock de leitura ao término da suspensão. O código detecta se entre as linhas 13 e 17 houve alguma inserção na fila, colocando o estado novamente no estado de pronto.

A tabela 5.5 mostra os tempos médios (milissegundos) para o exemplo de produtor e consumidor, agora implementado em Lua. Assim como em C, estamos mostrando o caso sem concorrência (SC) e com concorrência (CC). Comparando com a tabela 5.3 de comunicação em C, podemos notar um aumento no tempo de troca de mensagem. Após alguns teste, notamos que esse aumento é decorrente da

```
1 function send(state)
2   writelock.lock(state.lock)
3   state.queue.push(msg)
4   if state.status == BLOCKED then
5      state.status = READY
6      resume(proc)
7   end
8   writelock.release()
9 end
10
11 function receive(state)
12   while true do
13     message msg = state.queue.pop_if_present()
14     if msg then
15       return msg
16     end
17     readlock.lock(state.lock)
18     suspend(state)
19   end
20 end
```

Figura 5.3: Pseudo-código do escalomentodo do processo Lua.

incorporação de string, dado que Lua faz a cópia da string para e ainda calcula um valor *hash* dela.

A tabela 5.6 apresenta os tempos médios para 10 produtores e um consumidor implementados em Lua — comparável com a tabela 5.4 de C. Em ambos os casos, cada um deles está sendo executado em um processo Lua diferente e há o mesmo número de threads e processos Lua, para que todos possam ser executados.

Mensagem	**Clássico (SC)**	**Lock-free (SC)**	**Clássico (CC)**	**Lock-free (CC)**
64 bytes	91,18	93,71	155,03	101,41
128 bytes	114,37	118,51	156,43	125,93
256 bytes	148,33	150,49	185,06	170,55
512 bytes	214,24	216,74	251,54	240,13
1.024 bytes	344,88	346,85	399,20	387,52
2.048 bytes	609,49	611,90	702,23	686,68
4.096 bytes	1.133,17	1.135,89	1.372,85	1.356,17

Tabela 5.5: Tempos médios, em milissegundos, de produtor e consumidor em Lua.

Além dos testes de produtor/consumidor, adicionamos um teste que força a suspensão com mais freqüência. Esse teste consiste de um *ping-pong* entre dois processos Lua, onde o primeiro envia uma mensagem ao segundo e espera que a mesma mensagem seja enviada de volta. O programa realiza 50.000 vezes esse procedimento (totalizando 100.000 mensagens trocadas) e a média de 10 execuções consecutivas é mostrada na tabela 5.7.

É interessante notar no exemplo sem concorrência da tabela 5.5 e nesse último

Mensagem	Clássico	Lock-free
64 bytes	164,65	105,42
128 bytes	160,31	122,71
256 bytes	184,17	164,51
512 bytes	246,83	240,90
1.024 bytes	402,80	391,06
2.048 bytes	704,23	691,26
4.096 bytes	1.366,17	1.366,67

Tabela 5.6: Tempos médios, em milissegundos, de 10 produtores e um consumidor em Lua.

Mensagem	Clássico	Lock-free
64 bytes	182,77	155,85
128 bytes	202,37	171,88
256 bytes	227,44	199,54
512 bytes	180,75	174,51
1.024 bytes	188,68	175,06
2.048 bytes	180,54	185,59
4.096 bytes	194,90	186,14

Tabela 5.7: Tempos médios, em milissegundos, do *ping-pong* entre dois processos Lua.

exemplo que o modelo clássico de fila obteve um desempenho um pouco melhor do que as filas lock-free. Isso se deve ao peso do lock dos escritores (sempre realizado) que é maior que o lock da biblioteca POSIX Thread usado na fila clássica. A tabela 5.8 mostra o tempo médio para adquirir e liberar os diferentes tipos de locks. A fila clássica necessita adquirir e liberar o lock para inserir e remover a mensagem da fila. Nossa implementação utilizando lock-free adquire e libera o lock de leitura da biblioteca TBB para inserir a mensagem na fila, mas não utiliza lock para tentar retirar mensagens da fila (o lock de escrita só é usado se não houver mensagem a ser retirada, mas no teste sem concorrência sempre há mensagem). Dessa forma, mesmo utilizando o lock duas vezes, a fila clássica acaba tendo vantagem pois o tempo de adquirir e liberar o seu lock é menos da metade do tempo do lock de leitura utilizado pela fila lock-free.

Por outro lado, ao analisarmos o caso da concorrência, a diferença é maior entre as duas filas, com vantagem para a fila lock-free — principalmente com tamanhos menores de mensagem. Isso se deve ao fato de que o lock de leitura permite que várias threads façam a inserção ao mesmo tempo. Além disso, não há lock para a remoção de mensagem, aumentando ainda mais o acesso concorrente à fila, enquanto que o acesso à fila clássica é seqüêncial.

Dada a análise de desempenho de lock-free até o momento, decidimos continuar nossa implementação inter-processos apenas com esse modelo de fila.

Tipo de Lock	Tempo
Lock do POSIX Thread	3,793 ms
Lock de leitura do TBB	8,894 ms
Lock de escrita do TBB	4,745 ms

Tabela 5.8: Tempos médios dos locks, em milissegundos.

Modelo Implementado

Descrevemos agora com mais detalhes o modelo implementado para a comunicação intra-processo. Iniciamos com duas funções básicas para recebimento das mensagens. A função `ccr.tryreceive` tenta remover uma mensagem da fila, retornando a string enviada para o processo. Caso não haja nenhuma informação, ela retorna o valor `nil`. Essa operação não é bloqueante, diferentemente de `ccr.recv` (em geral utilizada internamente). Esta função não retorna enquanto não houver uma mensagem para ser consumida. Isso bloqueia o andamento do programa Lua e a própria thread.

Empregando essas duas funções, podemos construir então a função mais geral `ccr.receive` que é utilizada na construção do loop de eventos. Essa função é implementada em código Lua e tem por objetivo suspender a execução do processo Lua caso não haja eventos. O código da função é mostrado na figura 5.4, e ele introduz novas funções que devem ser explicadas.

```
1  function ccr.receive()
2     if coroutine.running() or ccr.ismain then
3        return ccr.recv()
4     else
5        while true do
6           local str = ccr.tryreceive()
7           if str then
8              return str
9           else
10             ccr.yield()
11          end
12       end
13    end
14 end
```

Figura 5.4: Implementação da função de recebimento de eventos.

A implementação de `ccr.receive` trata alguns casos especiais no nosso modelo de escalonamento de threads e processos. Primeiramente vamos explicar o laço da linha 5 que é a parte mais simples. Nele, é verificado se há um evento na fila e, caso haja, é retornado na linha 8. Se a fila está vazia, a função `ccr.yield` é invocada, sendo responsável por suspender a execução do processo Lua atual e deixar a thread livre para executar outro processo. A implementação também lida

com o caso onde mensagens podem ter sido colocadas na fila desde a chamada à `ccr.tryreceive`.

Para implementarmos a suspensão e retomada dos processos Lua, utilizamos a API C de co-rotinas de Lua. As funções C `lua_yield` e `lua_resume`, respectivamente, suspende a execução e retoma o processamento de uma co-rotina. Lua oferece a facilidade de disparar o próprio programa principal como sendo uma co-rotina, em vez deste criar uma co-rotina para então ser executada. Podemos carregar um código principal Lua e chamar `lua_resume` para iniciar sua execução, o que permite invocar `coroutine.yield` (em Lua) ou `lua_yield` (em C) para suspendê-lo como se fosse uma co-rotina. A figura 5.5 mostra um exemplo de uso dessas funções.

Um programa Lua definido nas linhas 3, 4 e 5 é carregado em um estado Lua (linha 11). Na linha 13, a chamada `lua_resume` inicia a execução desse programa simples, o qual exibe na tela a mensagem *Suspendendo* e, em seguida, suspende a execução com `coroutine.yield` (um módulo C poderia chamar `lua_yield`). O controle volta para a chamada `lua_resume` que retorna um código indicando o estado da execução. O programa C exibe uma mensagem referente a esse estado e termina. Se desejássemos retomar a execução do programa Lua, bastaria invocar `lua_resume` novamente.

De volta à implementação de `ccr.receive`, temos duas situações em que `ccr.yield` não pode ser usada: (i) dentro de uma co-rotina e (ii) em um estado que não foi criado com `ccr.spawn`.

Se fizermos uma chamada a `ccr.yield` dentro de uma co-rotina, a função irá configurar a nossa estrutura de controle para o bloqueio. No entanto, ao realizar a chamada `lua_yield`, `ccr.yield` suspenderá apenas a co-rotina e não todo o processo Lua. Como o controle não retorna ao lado C de nossa biblioteca para a finalização do bloqueio, isso causará inconsistência e comprometerá a execução de todo o programa.

O caso (ii) está relacionado com à nossa decisão de provermos a funcinalidade via biblioteca auxiliar. Precisamos que os programas Lua carreguem a biblioteca `ccr`. Se um programa executando em um estado que não foi lançado com `ccr.spawn`, não poderemos empregar o mecanismo suspensão, pois não temos garantias de que `lua_resume` foi usado para iniciar o programa. Por exemplo, o interpretador padrão utiliza `lua_pcall` para executar os programas Lua. Dessa forma, devemos fazer distinção entre esse primeiro estado (que chamaremos de *estado principal*) e os demais estados controlados por `ccr`, para termos a certeza de que o escalomento funcionará adequadamente.

Os programas Lua podem consultar a variável booleana `ccr.ismain` para verificar se eles estão sendo executados em um estado princiapal. A imlementação

```
1 int main(int argc, char* argv[])
2 {
3    char code[] = "print('Suspendendo')\n"
4                  "coroutine.yield()\n"
5                  "print('Retomado')\n";
6    /* Novo ambiente de execução Lua */
7    lua_State *L = luaL_newstate();
8    /* Carrega as bibliotecas padrão */
9    luaL_openlibs(L);
10   /* Carrega o script Lua */
11   luaL_loadstring(L, code);
12   /* Inicia a execução do script */
13   switch (lua_resume(L, 0)) {
14   case 0:
15     printf("Término do script Lua.\n");
16     break;
17   case LUA_YIELD:
18     printf("Execução suspensa.\n");
19     break;
20   default:
21     printf("Erro na execução\n");
22     break;
23   }
24   lua_close(L);
25   return EXIT_SUCCESS;
26 }
```

Figura 5.5: Exemplo da API C de co-rotinas que é usada na construção do escalonador do módulo `ccr`.

de `ccr.receive` verifica essa variável (linha 2 na figura 5.4) para evitar a chamada de `ccr.yield` e conseqüentemente os problemas descritos. A função `ccr.receive` invoca a função `ccr.recv` para manter o programa esperando uma mensagem da fila.

A primitiva `ccr.send` é responsável por enviar mensagem para os estados. Caso o estado destinatário esteja bloqueado, `ccr.send` o move para a fila de estados prontos. Ela recebe o identificador e a string a ser enviada. Essa função está acessível tanto de C como Lua para que seja possível criar outras bibliotecas geradoras de eventos.

5.2 Integração Inter-processos

Tendo implementada a biblioteca básica de comunicação, devemos agora integrá-la com o modelo do ALua, fornecendo assim uma arquitetura de eventos e multitarefa com threads e processos.

A infra-estrutura do ALua é baseada em uma rede de *daemons* e processos (executando código Lua), como mostra a figura 5.6. Os daemons são responsáveis por interligar máquinas diferentes e de criar novos processos. Eles também cum-

prem o papel de ponto de entrada "bem conhecido" nas máquinas e de multiplexação da comunicação entre as máquinas, reduzindo o número total de conexões.

A comunicação no ALua se dá através de canais TCP, o que permite uma interação tanto dentro da mesma máquina, bem como remotamente. Por padrão, toda a comunicação passa pelos daemons, mas nada impede o desenvolvedor de criar suas conexões diretamente entre os processos caso seja necessário uma comunicação sem intermediários. Nesse caso, o desenvolvedor fica totalmente responsável pelas conexões criadas.

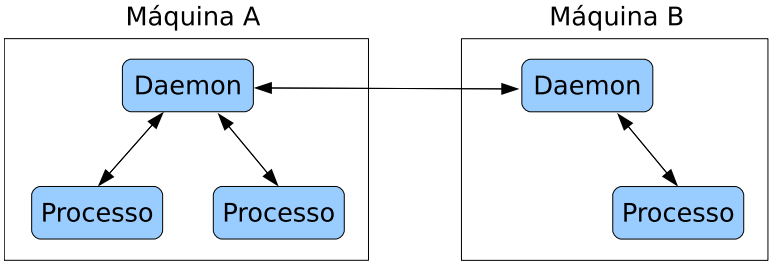

Figura 5.6: Arquitetura do ALua.

Com a integração da biblioteca `ccr`, os processos da arquitetura do ALua, que eram processos do sistema operacional executando um programa Lua, passam a conter vários processos Lua. Esses processos Lua se comunicam internamente pela fila de mensagens da biblioteca `ccr` e compartilham o canal TCP. A figura 5.7 apresenta a nova arquitetura do processo.

Novos processos são criados com a função `alua.spawn`, que recebe uma string contendo o código Lua e uma função opcional de callback, a qual é chamada para sinalizar que o novo processo foi criado. Ao estendermos o ALua, modificamos a função `alua.spawn` para suportar mais um parâmetro booleano, indicando se o novo processo Lua será criado compartilhando o mesmo processo do sistema operacional ou em um novo.

Cada um dos processos Lua tem um identificador, o qual é uma string única que indica a localização do processo no sistema. É através dele que o sistema sabe se deve enviar o evento diretamente pela fila de mensagens ou encaminhá-lo ao daemon para o roteamento.

Podemos ver, na figura 5.7, a existência de dois processos Lua diferentes: normal e despachante. Com a utilização de um canal externo compartilhado, necessitamos coordenar o envio e recebimento dos dados. O processo Lua despachante fica responsável pela tarefa de receber as informações e repassá-las para seu respectivo destinatário. Essa tarefa exige uma ação ativa, verificando constantemente junto ao sistema operacional se há informação no canal. Por isso, o despachante nunca pode ter sua execução suspensa na fila de mensagens interna como os demais. Nesse caso, utilizamos a primitiva `ccr.tryreceive` para construir o loop de

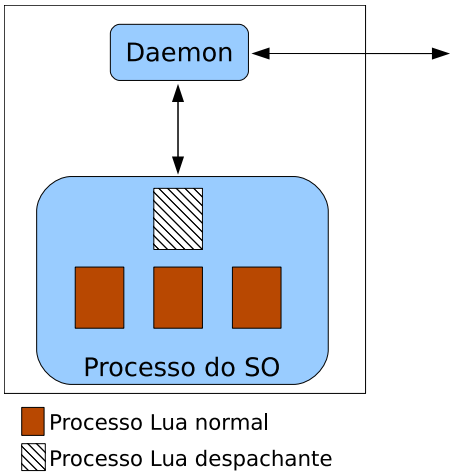

Figura 5.7: ALua com diversos processos Lua.

eventos do despachante, juntamente com um `select` não bloqueante para verificação dos sockets. A figura 5.8 apresenta o pseudo-código do novo loop de eventos do ALua.

```
function alua.loop()
  if dispatcher then
    while true do
      -- Verifica se há mensagem na fila
      msg = ccr.tryreceive()
      if msg then
        process(msg)
      end
      -- Verifica se há mensagem do daemon
      ready = net.select({daemonsock})
      if ready[1] then
        msg = net.receive(ready[1])
        if msg then
          process(msg)
        end
      end
    end
  else
    -- Verifica apenas a fila de mensagens
    while true do
      msg = ccr.receive()
      process(msg)
    end
  end
end
```

Figura 5.8: Novo loop de eventos do ALua.

Em relação ao envio de mensagens pelo canal TCP, os processos Lua não precisam de intermediário. Por outro lado, devemos garantir que isso seja feito de forma ordenada. O envio é feito pela primitiva `alua.send`, que recebe como argumentos o identificador do destinatário e a string. Se o destinatário está localizado no

mesmo processo do sistema operacional, a mensagem é enviada diretamente, caso contrário, `alua.send` repassa a mensagem para o canal externo. Utilizamos um lock para envolver a função de envio externo, assegurando a coordenação e a multiplexação correta das mensagens.

Para finalizar a descrição de nosso sistema, apresentamos um programa exemplo *ping-pong* na figura 5.9 para ilustrar que a API do ALua não sofreu grandes modificações, permanecendo praticamente a mesma. O programa principal utiliza a função `alua.init` para se conectar com o daemon. Ela funciona de maneira assíncrona e retorna imediatamente, fazendo com que o programa vá para a execução do loop de eventos. Após a inicialização ter sido completada, a função de callback `inited` é executada, a qual cria um novo processo Lua, no mesmo processo do sistema operacional, contendo o código `code`. Finalmente, a callback `start` é executada e os dois iniciam as chamadas de um para o outro, alternadamente.

```
require("alua")

local code = [[
  -- 'alua.id' contém o próprio identificador do processo
  evt_ping = string.format("ping(%q)", alua.id)
  function pong(dst)
    print("pong")
    alua.send(dst, evt_ping)
  end
]]

function ping(dst)
  print("ping")
  alua.send(dst, evt_pong)
end

function start(reply)
  -- 'reply.id' contém o identificador do processo criado
  ping(reply.id)
end

function inited()
  -- 'alua.id' contém o próprio identificador do processo
  evt_pong = string.format("pong(%q)", alua.id)
  alua.spawn(code, start, true)
end

alua.init{addr = "127.0.0.1", port = 8888, callback = inited}
alua.loop()
```

Figura 5.9: Exemplo *ping-pong* no ALua.

Nesse exemplo, como o programa é restrito a um processo do sistema operacional, não haveria a necessidade de comunicação externa e, assim, do daemon. Resolvemos simplificar a arquitetura para que esse tipo de programa não necessite

de conexão com o daemon. Para isso, basta omitir na função `alua.init` o endereço de conexão TCP.

Ao mantermos a mesma API do ALua original, juntamente com a visão de processos independentes, podemos utilizar os mesmos mecanismos comunicação RPC síncrono e assíncrono discutidos no capítulo 3. Isso também nos permite usar as abstrações de coordenações descritas. De fato, exploramos esses mecanismos em um dos casos de uso apresentados no próximo capítulo.

5.3 Desempenho

Nesta seção, apresentamos o desempenho da nossa reimplementação do ALua, de acordo com o modelo proposto no capítulo 4. Realizamos os testes de eventos entre processos Lua dentro de um mesmo processo do sistema operacional, comunicação intra-processo, e em processos diferentes do sistema operacional, comunicação inter-processo.

Usaremos novamente o exemplo de produtor e consumidor para verificarmos o desempenho na troca de eventos. Os testes anteriores avaliaram as partes básicas de comunicação, como a transmissão de uma string via fila ou conexão TCP. No entanto, com o sistema ALua completo, temos questões como a decisão de roteamento da mensagem, mensagens de controle e invocação do tratador.

O produtor gera 100.000 eventos de diferentes tamanhos, os quais serão tratados pelo consumidor. O tratador dos eventos neste caso não realizará nenhuma ação, permitindo a medição apenas do tempo gasto na infra-estrutura. O tempo é medido a partir do primeiro envio até o recebimento de todas as mensagens. Para testarmos mensagens com diferentes tamanhos, enviamos eventos que contêm um parâmetro string. Variamos o tamanho dessa string para obtermos o tamanho de mensagem desejado.

Realizamos também um teste similar com Erlang (versão 5.5.5), para efeito de comparação do tempo de comunicação entre processos. Na comunicação inter-processo, utilizamos dois processos leves como produtor e consumidor. Na comunicação intra-processos, disparamos dois interpretadores Erlang em terminais diferentes e empregamos a primitiva `spawn` para disparar um processo remoto e então fazer a comunicação com o mesmo, utilizando o operador "`!`". Para efeito de comparação, adicionamos uma segunda forma de comunicação inter-processo para Erlang, utilizando o módulo `rpc` disponível na biblioteca padrão. Assim como o produtor no ALua, o chamador passa um parâmetro variável.

Os tempos medidos estão na tabela 5.9, em segundos. As colunas "Intra" se referem à comunicação intra-processo e "Inter" à comunicação inter-processos. Na comunicação inter-processo do ALua, medimos o tempo considerando primeiro o

despachante como consumidor (coluna "Inter-Desp") e depois um outro processo qualquer como consumidor ("Inter"). Pela nossa arquitetura, o despachante é o intermediário no recebimento dos eventos, o que agrega um tempo de repasse da mensagem além da comunicação TCP.

Mensagem	**ALua**			**Erlang**		
	Intra	Inter-Desp	Inter	Intra	Inter-Spawn	Inter-RPC
64 bytes	2,171	7,698	9,345	0,079	0,624	8,462
128 bytes	2,392	7,980	9,579	0,112	0,730	8,601
256 bytes	2,821	8,721	10,544	0,434	1,240	9,039
512 bytes	3,643	10,144	11,660	0,546	3,177	9,130
1.024 bytes	5,294	12,151	14,057	2,760	4,101	9,831
2.048 bytes	8,458	16,349	18,566	2,427	7,291	14,818
4.096 bytes	14,771	24,638	25,938	4,206	9,212	20,324

Tabela 5.9: Tempos médios, em segundos, da comunicação entre processos no ALua e Erlang.

Erlang possui um desempenho melhor para a comunicação entre processos, principalmente para mensagens menores. Com a característica de valores imutáveis da linguagem, é possível otimizar a passagem interna de mensagem, compartilhando diretamente os valores entre os processos, sem o risco de inconsistência (Johansson et al., 2002).

Em nosso modelo, os processos Lua são independentes e não podemos utilizar uma estratégia de compartilhamento de valores da linguagem. Por isso, pagamos um preço maior para transferir os valores de um processo Lua para outro. E além do evento em si, temos mais informações de controle do ALua que devem ser serializadas e enviadas. De fato, as mensagens são internamente montadas como tabelas, adicionando informações de controle. Elas são então serializadas em strings e transformadas novamente em elemento Lua no destinatário usando a função `loadstring`. Esse mecanismo pode ter dado um peso na comunicação.

Na comunicação inter-processos, podemos notar a interferência do despachante na comunicação fim-a-fim do ALua. Além disso, pela arquitetura do ALua, as mensagens inter-processos são roteadas pelo daemon, o que adiciona mais uma indireção.

5.4 Consumo de Memória

Realizamos a medida de consumo de memória para analisarmos o impacto da criação de vários processos Lua. Para efetuar essa medida, utilizamos o comando `pmap` que mostra como a memória virtual usada pelo programa está sendo mapeada e o consumo total. Em vez de empregarmos o interpretador padrão de Lua para

lançar os testes, escrevemos o nosso próprio programa C responsável por iniciar o script Lua de criação dos processos Lua, para termos mais controle sobre o que está sendo criado e sobre pausas para realizar as medidas de memória.

A tabela 5.10 mostra as medidas, em kilobytes (Kb), com diferentes números de processos Lua criados. Essa tabela mostra o consumo total e exclue a memória alocada para as threads, contabilizando apenas o programa principal C e os processos Lua instaciados com suas depedências. O apêndice B mostra o resultado detalhado do `pmap`. As medidas foram feitas em uma máquina equipada com processador Pentium Core 2 Quad (quatro núcleos, cada um com 2,4GHz), executando Linux 64 bits, kernel 2.6.24

A coluna *Memória Total* indica a quantidade total de memória virtual consumida pelo programa após ter instanciado os processos. A linha com um processo Lua é nossa base de comparação. Ela engloba o mínimo para iniciar um programa que utilizará o módulo ALua juntamente com todos os módulos padrão de Lua. A coluna *Acréscimo* representa o aumento de uso de memória virtual que o programa teve sobre a nossa base de um processo Lua. Sobre esse ganho, calculamos na última coluna o consumo por processo Lua.

Processos Lua	**Memória Total**	**Acréscimo**	**Acréscimo por Processo**
1	4.500 Kb	–	–
100	6.692 Kb	2.112 Kb	21,12 Kb
1.000	26.620 Kb	22.040 Kb	22,04 Kb
10.000	231.320 Kb	226.740 Kb	22,67 Kb

Tabela 5.10: Consumo de memória dos processos Lua.

Podemos notar que cada processo Lua tem um custo médio em torno de 22 kilobytes. Se compararmos esse valor com o custo de uma thread, que na configuração do Linux instalado consome 10.240 kilobytes de memória virtual para a pilha, notamos que o custo de um processo Lua é aproximadamente 465 vezes menor. No entanto, o modelo é baseado em threads, dessa forma, um número adequando de threads deverá ser criado.