Bibliografia

[1] ADYA, A.; HOWELL, J.; THEIMER, M.; BOLOSKY, W. J. ; DOUCER, J. R.. **Cooperative Task Management without Manual Stack Management**. In: PROCEEDINGS OF USENIX ANNUAL TECHNICAL CONFERENCE, Monterey, CA, June 2002. USENIX.

[2] ANDREWS, G. R.. **Foundations of Multithreaded, Parallel and Distributed Programming**. Addison-Wesley, 2000.

[3] ARCHER, T.; WHITECHAPEL, A.. **Inside C#**. Microsoft Press, 2002.

[4] ATKINSON, R.; LISKOV, B. ; SCHELFLER, R.. **Aspects of Implementing CLU**. In: PROCEEDINGS OF THE ACM 1978 ANNUAL CONFERENCE, p. 123–129, Oct. 1978.

[5] BEHREN, R.; CONDIT, J. ; BREWER, E.. **Why Events are a Bad Idea (for high-concurrency servers)**. In: PROCEEDINGS OF THE 9TH WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS (HOTOS IX), Lihue, HI, May 2003.

[6] BEN-ARI, M.. **Principles of Concurrent and Distributed Programming**. Benjamin Cummings, 1990.

[7] BENTON, N.; CARDELLI, L. ; FOURNET, C.. **Modern Concurrency Abstractions for C#**. In: PROCEEDINGS OF THE SIXTEENTH EUROPEAN CONFERENCE ON OO PROGRAMMING ECOOP 2002, Spain, June 2002.

[8] BIRTWISTLE, G.; DAHL, O.-J.; MYHRHAUG, B. ; NYGAARD, K.. **Simula Begin**. Studentlitteratur, Sweden, 1980.

[9] BIRREL, A. D.. **An introduction to programming with threads**. Technical report 35, Digital Systems Research Center, Jan. 1989.

[10] BRUGGEMAN, C.; WADDELL, O. ; DYBVIG, R.. **Representing Control in the Presence of One-Shot Continuations**. In: PROCEEDINGS OF THE ACM SIGPLAN'96 CONF. ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI), p. 99–107, Philadelphia, PA, May 1996. ACM. SIGPLAN Notices 31(5).

[11] CARZANIGA, A.; ROSENBLUM, D. ; WOLF, A.. **Design and evaluations of a wide-area event notification service**. ACM Transactions on Computer Systems, 19(3):332–383, 2001.

[12] CLINGER, W.; HARTHEIMER, A. ; OST, E.. **Implementation strategies for continuations**. In: PROCEEDINGS OF THE 1988 ACM CONFERENCE ON LISP AND FUNCTIONAL PROGRAMMING, p. 124–131, Snowbird, Utah, 1988.

[13] CLOCKSIN, W.; MELLISH, C.. **Programming in Prolog**. Springer-Verlag, 1981.

[14] CONWAY, M.. **Design of a separable transition-diagram compiler**. Communications of the ACM, 6(7):396–408, July 1963.

[15] CONWAY, D.. **RFC 31: Subroutines: Co-routines**, 2000. `http://dev.perl.org/perl6/rfc/31.html`.

[16] DABEK, F.; ZELDOVICH, N.; KAASHOEK, F.; MAZIERES, D. ; MORRIS, R.. **Event-driven Programming for Robust Software**. In: PROCEEDINGS OF THE 10TH ACM SIGOPS EUROPEAN WORKSHOP, Sept. 2002.

[17] DAHL, O.-J.; DIJKSTRA, E. W. ; HOARE, C. A. R.. **Hierarchical program structures**. In: STRUCTURED PROGRAMMING, p. 175–220. Academic Press, London, England, 1972.

[18] DANVY, O.; FILINSKI, A.. **Abstracting control**. In: PROCEEDINGS OF THE 1990 ACM CONFERENCE ON LISP AND FUNCTIONAL PROGRAMMING, p. 151–160, Nice, France, June 1990. ACM.

[19] DIJKSTRA, E. W.. **Cooperating Sequential Processes**. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, Jan. 1965.

[20] DIJKSTRA, E. W.. **The Structure of the THE Multiprogramming System**. Communications of the ACM, 11(5):341–346, 1968.

[21] DYBVIG, R.; HIEB, R.. **Engines from continuations**. Computer Languages, 14(2):109–123, 1989.

[22] FELLEISEN, M.. **Transliterating Prolog into Scheme**. Technical Report 182, Indiana University, Bloomington, IN, 1985.

[23] FELLEISEN, M.; FRIEDMAN, D.. **Control operators, the secd-machine, and the λ-calculus**. In: Wirsing, M., editor, FORMAL DESCRIPTION OF PROGRAMMING CONCEPTS-III, p. 193–217. North-Holland, 1986.

[24] FELLEISEN, M.. **The theory and practice of first-class prompts**. In: PROCEEDINGS OF THE 15TH ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES POPL'88, p. 180–190, San Diego, CA, Jan. 1988. ACM.

[25] FELLEISEN, M.. **On the expressive power of programming languages**. In: PROCEEDINGS OF 3RD EUROPEAN SYMPOSIUM ON PROGRAMMING ESOP'90, p. 134–151, Copenhagen, Denmark, May 1990.

[26] FELLEISEN, M.. **Developing Interactive Web Programs**. In: ADVANCED FUNCTIONAL PROGRAMMING: 4TH INTERNATIONAL SCHOOL, AFP 2002, volumen 2638 de **Lecture Notes in Computer Science**, p. 100–128. Springer-Verlag, 2003.

[27] FRIEDMAN, D.; HAYNES, C. ; KOHLBECKER, E.. **Programming with continuations**. In: Pepper, P., editor, PROGRAM TRANSFORMATION AND PROGRAMMING ENVIRONMENTS, p. 263–274. Springer-Verlag, 1984.

[28] FRIEDMAN, D.; HAYNES, C.; KOHLBECKER, E. ; WAND, M.. **Scheme 84 Interim Reference Manual**. Technical report 153, Indiana University, Computer Science Department, 1985.

[29] FRIEDMAN, D.; WAND, M. ; HAYNES, C.. **Essentials of Programming Languages**. MIT Press, London, England, second edition, 2001.

[30] FUCHS, M.. **Escaping the event loop: an alternative control structure for multi-threaded GUIs**. In: Unger, C.; Bass, L., editors, Engineering for the HCI, p. 69–87. Chapman and Hall, 1996.

[31] GANZ, S.; FRIEDMAN, D. ; WAND, M.. **Trampolined Style**. In: PROCEEDINGS OF THE 4TH ACM SIGPLAN INTERNATIONAL

CONFERENCE ON FUNCTIONAL PROGRAMMING, p. 18–27, Paris, France, 1999.

[32] GELERNTER, D.; CARRIERO, N.. **Generative Communications in Linda**. ACM Transactions on Programming Languages and Systems, 7(1):80–112, 1985.

[33] GOLDBERG, A.; ROBSON, D.. **Smalltalk-80: the Language and its Implementation**. Addison-Wesley, 1983.

[34] GRAUNKE, P.; KRISHNAMURTI, S.; HOEVEN, S. V. D. ; FELLEISEN, M.. **Programming the Web with High-Level Programming Languages**. In: EUROPEAN SYMPOSIUM ON PROGRAMMING, ESOP'2001, Apr. 2001.

[35] GRISWOLD, R.. **The Evaluation of Expressions in Icom**. ACM Transactions on Programming Languages and Systems, 4(4):563–584, 1982.

[36] GRISWOLD, R.; GRISWOLD, M.. **The Icon Programming Language**. Prentice-Hall, New Jersey, NJ, 1983. 3rd Edition, Peer to Peer Communications, 2000.

[37] GRUNE, D.. **A View of Coroutines**. ACM SIGPLAN Notices, 12(7):75–81, 1977.

[38] HANSEN, P. B.. **The Nucleus of a Multiprogramming System**. Communications of the ACM, 13(4):238–241 and 250, 1970.

[39] HANSEN, P. B.. **Operating System Principles**. Prentice-Hall, 1973.

[40] HARPER, R.; DUBA, B.; HARPER, R. ; MACQUEEN, D.. **Typing first-class continuations in ML**. In: PROCEEDINGS OF THE 18TH ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES POPL'91, p. 163–173, Orlando, FL, Jan. 1991. ACM.

[41] HAYNES, C. T.; FRIEDMAN, D. ; WAND, M.. **Obtaining coroutines with continuations**. Computer Languages, 11(3/4):143–153, 1986.

[42] HAYNES, C. T.. **Logic continuations**. J. Logic Programming, 4:157–176, 1987.

[43] HELSGAUN, K.. **A Portable C++ Library for Coroutine Sequencing**. Writings on Computer Science, Department of Computer Science, Roskilde University, Roskilde, Denmark, 1999.

[44] HEWITT, C.. **Viewing control structures as patterns of passing messages**. Artificial Intelligence, 8, 1977.

[45] HIEB, R.; DYBVIG, R. ; BRUGGEMAN, C.. **Representing Control in the Presence of First-Class Continuations**. In: PROCEEDINGS OF THE ACM SIGPLAN'90 CONF. ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI), p. 66–77, White Plains, NY, June 1990. ACM. SIGPLAN Notices 25(6).

[46] HIEB, R.; DYBVIG, R. ; ANDERSON III, C. W.. **Subcontinuations**. Lisp and Symbolic Computation, 7(1):83–110, 1994.

[47] HU, J.; PYARALI, I. ; SCHMIDT, D. C.. **Applying the Proactor Pattern to High-Performance Web Servers**. In: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS, IASTED, Las Vegas, Nevada, Oct. 1998.

[48] IERUSALIMSCHY, R.; FIGUEIREDO, L. ; CELES, W.. **Lua — an extensible extension language**. Software: Practice & Experience, 26(6):635–652, June 1996.

[49] IERUSALIMSCHY, R.. **Programming in Lua**. Lua.org, ISBN 85-903798-1-7, Rio de Janeiro, Brazil, 2003.

[50] JOHNSON, G.; DUGGAN, D.. **Stores and partial continuations as first-class objects in a language and its environment**. In: PROCEEDINGS OF THE 15TH ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES POPL'88, San Diego, CA, Jan. 1988. ACM.

[51] KELSEY, R.; CLINGER, W. ; REES, J.. **Revised^5 report on the algorithmic language Scheme**. ACM SIGPLAN Notices, 33(9):26–76, Sept. 1998.

[52] KNUTH, D. E.. **The Art of Computer Programming, Volume 1, Fundamental Algorithms**. Addison-Wesley, Reading, MA, 1968. 3rd Edition, 1997.

[53] KUMAR, S.; BRUGGEMAN, C. ; DYBVIG, R.. **Threads yield continuations**. Lisp and Symbolic Computation, 10(3):223–236, 1998.

[54] LARUS, J.; PARKES, M.. **Using Cohort Scheduling to Enhance Server Performance**. In: USENIX ANNUAL TECHNICAL CONFERENCE, June 2002.

[55] LAUER, H. C.; NEEDHAM, R. M.. **On the Duality of Operating System Structures**. In: PROC. SECOND INTERNATIONAL SYMPOSIUM ON OPERATING SYSTEMS, IRIA, Oct. 1978. Reprinted in Operating Systems Review, 13(2), pages 3–19, 1979.

[56] LEA, D.. **Concurrent Programming in Java, Design Principles and Patterns**. Addison-Wesley, Reading, MA, second edition, 2000.

[57] LEAL, M.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **LuaTS - a reactive event-driven tuple space**. J.UCS - Journal of Universal Computer Science, 9(8):730–744, 2003.

[58] DE LIMA, M. J. D.. **ORFEO: Programação Distribuída Orientada a Eventos com Funções e Continuações como Valores de Primeira Classe**. PhD thesis, Departamento de Informática, PUC–Rio, 2000.

[59] LISKOV, B.; SNYDER, A.; ATKINSON, R. ; SCHAFFERT, C.. **Abstraction mechanisms in CLU**. Communications of the ACM, 20(8):564–576, Aug. 1977.

[60] LISKOV, B.; MOSS, E.; SNYDER, A.; ATKINSON, R.; SCHAFFERT, C.; BLOOM, T. ; SCHEIFLER, R.. **CLU Reference Manual**. Springer-Verlag, New York, NY, 1984.

[61] MADSEN, O.; M.-PEDERSEN, B. ; NYGAARD, K.. **Object-oriented programing in the BETA programming language**. ACM Press/Addison-Wesley, New York, NY, 1993.

[62] MARLIN, C. D.. **Coroutines: A Programming Methodology, a Language Design and an Implementation**. LNCS 95, Springer-Verlag, 1980.

[63] MARTELLI, A.. **Python in a Nutshell**. O'Reilly, 2003.

[64] MOODY, K.; RICHARDS, M.. **A coroutine mechanism for BCPL**. Software: Practice & Experience, 10(10):765–771, Oct. 1980.

[65] MOURA, A. L.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Coroutines in Lua**. In: 8TH BRAZILIAN SYMPOSIUM OF PROGAMMING LANGUAGES (SBLP), p. 89–101, Niteroi, RJ, Brazil, May 2004. SBC.

[66] MURER, S.; OMOHUNDRO, S.; STOUTAMIRE, D. ; SZYPERSKI, C.. **Iteration abstraction in Sather**. ACM Transactions on Progamming Languages and Systems, 18(1):1–15, Jan. 1996.

[67] NICHOLS, B.; BUTTLAR, D. ; FARRELL, J. P.. **Pthreads Programming**. O'Reilly & Associates, 1996.

[68] OUSTERHOUT, J.. **Why threads are a bad idea (for most purposes)**. In: USENIX TECHNICAL CONFERENCE, Austin, Texas, Jan. 1996. Invited Talk.

[69] PAULI, W.; SOFFA, M. L.. **Coroutine behaviour and implementation**. Software: Practice & Experience, 10(3):189–204, Mar. 1980.

[70] QUEINNEC, C.; SERPETTE, B.. **A dynamic extent control operator for partial continuations**. In: PROCEEDINGS OF THE 18TH ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES POPL'91, p. 174–184, Orlando, FL, Jan. 1991. ACM.

[71] QUEINNEC, C.. **A library of higher-level control operators**. ACM SIGPLAN Lisp Pointers, 6(4):11–26, Oct. 1993.

[72] QUEINNEC, C.. **The influence of browsers on evaluators, or continuations to program web servers**. In: INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING ICFP'2000, p. 23–33, Montreal, Canada, Sept. 2000.

[73] QUEINNEC, C.. **Inverting back the inversion of control, or Continuations versus page-centric programming**. ACM SIGPLAN Notices, 38(2):57–64, Feb. 2003.

[74] REYNOLDS, J. C.. **The Discoveries of Continuations**. Lisp and Symbolic Computation, 6(3/4):233–247, 1993.

[75] RICHTER, J.. **Advanced Windows**. Microsoft Press, Redmond, WA, third edition, 1997.

[76] ROSSETTO, S.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Abstrações para o desenvolvimento de aplicações distribuídas em ambientes com mobilidade**. In: 8TH BRAZILIAN SYMPOSIUM

OF PROGAMMING LANGUAGES (SBLP), p. 143–156, Niteroi, RJ, Brazil, May 2004. SBC.

[77] SCHMIDT, D.; STAL, M.; ROHNERT, H. ; BUSCHMANN, F.. **Pattern-Oriented Software Architecture, volume 2, Patterns for Concurrent and Networked Objects**. John Wiley & Sons, 2000.

[78] SCHEMENAUER, N.; PETERS, T. ; HETLAND, M.. **PEP 255 Simple Generators**, 2001. `http://www.python.org/peps/pep-0255.html`.

[79] SITARAM, D.. **Handling control**. In: PROCEEDINGS OF THE ACM SIGPLAN'93 CONF. ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI), Albuquerque, NM, June 1993. ACM. SIGPLAN Notices 28(6).

[80] SITARAM, D.. **Models of Control and Their Implications for Progamming Language Design**. PhD thesis, Rice University, Apr. 1994.

[81] SPRINGER, G.; FRIEDMAN, D. P.. **Scheme and The Art of Programming**. McGraw–Hill, 1994.

[82] STRACHEY, C.; WADSWORTH, C.. **Continuations: a Mathematical Semantics for Handling Full Jumps**. Higher-Order and Symbolic Computation, 13(1/2):135–152, 2000.

[83] THOMAS, D.; HUNT, A.. **Programming Ruby: The Pragmatic Programmer's Guide**. Addison-Wesley, 2001.

[84] TISMER, C.. **Continuations and Stackless Python**. In: PROCEEDINGS OF THE 8TH INTERNATIONAL PYTHON CONFERENCE, Arlington, VA, Jan. 2000.

[85] TOERNIG, E.. **C Coroutines (coro library)**, 2000. `http://www.goron.de/~froese/coro/coro.html`.

[86] URURAHY, C.; RODRIGUEZ, N.. **ALua: An event-driven communication mechanism for parallel and distributed programming**. In: PROC. 12TH INTL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING SYSTEMS (PDCS'99), Fort Lauderdale, Florida, 1999.

[87] WALL, L.; CHRISTIANSEN, T. ; ORWANT, J.. **Programming Perl**. O'Reilly, third edition, 2000.

[88] WAND, M.. **Continuation-based multiprocessing**. In: PROCEEDINGS OF THE 1980 LISP CONFERENCE, p. 19–28, Stanford, CA, Aug. 1980. ACM.

[89] WELSH, M.; CULLER, D. ; BREWER, E.. **SEDA: An Architecture for Well-Conditioned, Scalable Internet Services**. In: PROCEEDINGS OF THE EIGHTEENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP-18), Canada, Oct. 2001.

[90] WIRTH, N.. **Programming in Modula-2**. Springer-Verlag, third, corrected edition, 1985.

[91] ERIGHTS.ORG. **Overview: Concurrency in E**, 2003. `http://www.erights.org/elib/concurrency/overview.html`.

A
Gerência cooperativa de tarefas com co-rotinas completas assimétricas

No Capítulo 6, mostramos uma implementação bastante simples de um ambiente de gerência cooperativa de tarefas baseado em co-rotinas completas assimétricas (Figura 6.1). O objetivo deste apêndice é complementar essa implementação, adicionando facilidades de sincronização e mecanismos que permitem obter um melhor desempenho.

A primeira modificação ao nosso ambiente é a adição de um *pool* de co-rotinas reutilizáveis para minimizar o custo de criação de novas tarefas. Nessa nova implementação, apresentada na Figura A.1, a tabela `pool` armazena as co-rotinas disponíveis para a execução de uma nova tarefa. A função `create_task` somente cria uma nova co-rotina se esse *pool* está vazio; caso contrário, uma co-rotina disponível é removida do *pool*. Ao final da execução de sua tarefa, uma co-rotina se suspende, ao invés de terminar; em sua próxima reativação (invocada por `create_task`), a co-rotina recebe a próxima tarefa a executar. A nova função `release_task` é chamada pelo escalonador quando uma co-rotina sinaliza o término de sua tarefa. Se o *pool* está em seu tamanho máximo, essa co-rotina é descartada. Se não, a referência para essa co-rotina (suspensa à espera de uma nova tarefa) é salva no *pool* de co-rotinas.

Nossa segunda modificação evita o bloqueio de uma aplicação quando uma tarefa solicita uma operação de entrada ou saída. Para isso, assumimos que essas solicitações são realizadas através de funções como a apresentada na Figura 6.2. Nesse caso, quando uma operação não se completa num tempo pré-determinado, a co-rotina em execução é suspensa, retornando ao escalonador uma referência para o recurso correspondente à operação invocada. A referência para o recurso é salva em uma tabela, para que o escalonador, quando possível, possar aguardar uma mudança de estado. Quando todas as tarefas vivas estão à espera de operações de entrada ou saída, o escalonador se bloqueia, invocando uma função auxiliar (`select`) que permite aguardar uma mudança de estado em algum dos recursos

```
MAX_POOL = ...  -- tamanho máximo do pool
pool  = {}      -- pool de co-rotinas
tasks = {}      -- lista de tarefas vivas

-- cria uma tarefa
function create_task(f)
  local co
  if table.getn(pool) == 0 then -- pool vazio
    co = coroutine.wrap(
           function()
             while true do
               -- executa tarefa
               f()
               -- suspende execução aguardando nova tarefa
               f = coroutine.yield("task ended")
               -- aguarda ativação pelo escalonador
               coroutine.yield()
             end
           end)
  else
    co = table.remove(pool)  -- retira co-rotina do pool
    co(f)                    -- e envia nova tarefa
  end
  table.insert(tasks, co)
end

-- libera uma co-rotina
function release_task(co)
  if table.getn(pool) < MAX_POOL then
    table.insert(pool, co)
  end
end

-- escalonador
function dispatcher()
  while true do
    local n = table.getn(tasks)
    if n == 0 then break end
    for i = 1, n do
      local res = tasks[i]()      -- reativa tarefa
      if res == "task ended" then -- tarefa terminou
        release_task(tasks[i])    -- libera co-rotina
        table.remove(tasks, i)    -- remove tarefa
        break
      end
    end
  end
end
```

Figura A.1: Gerência cooperativa de tarefas com pool de co-rotinas

```
function dispatcher()
  while true do
    local n = table.getn(tasks)
    if n == 0 then break end
    resources = {}                -- tabela de recursos
    for i = 1, n do
      local res = tasks[i]()
      if res == "task ended" then -- tarefa terminou
        release_task(tasks[i])
        table.remove(tasks, i)
        break
      elseif res ~= nil then      -- tarefa aguarda E/S
        table.insert(resources, res)
      end
    end

    -- todas as tarefas vivas aguardam E/S ?
    if table.getn(resources) == table.getn(tasks) then
      select(resources)
    end
  end
end
```

Figura A.2: Gerência cooperativa de tarefas com E/S não bloqueante

referenciados na tabela de recursos. A Figura A.2 mostra a adição desses procedimentos ao escalonador.

Para completar nosso ambiente de gerência cooperativa de tarefas, podemos adicionar alguns mecanismos básicos de sincronização. Em primeiro lugar, implementamos um mecanismo de exclusão mútua baseado no conceito de semáforos binários [19, 6]. Nessa implementação, apresentada na Figura A.3, um semáforo é representado por uma tabela que armazena o valor *s* do semáforo (0 ou 1) e uma tabela de co-rotinas que representa a fila de tarefas que aguardam a liberação do semáforo. A operação *P(s)* é implementada por uma função que suspende a co-rotina ativa caso o semáforo não esteja liberado, retornando ao escalonador um valor que indica que a co-rotina deve ser inserida na fila do semáforo especificado, e removida da lista de tarefas vivas. Caso contrário, o valor do semáforo é alterado, e a co-rotina prossegue sua execução. A operação *V(s)* é implementada por uma função que, após atualizar o valor do semáforo, verifica se existem tarefas que o aguardam, consultando a fila de espera correspondente. Se essa fila não está vazia, a primeira co-rotina é removida da fila de espera e reinserida na lista de tarefas vivas.

Essa implementação pode causar uma distribuição injusta de recursos,

```
-- cria um semáforo
function create_sem() return {s = 0, blocked = {}} end

-- aguarda um semáforo
function P(sem)
  while sem.s > 0 do coroutine.yield("blocked", sem) end
  sem.s = 1
end

-- libera um semáforo
function V(sem)
  sem.s = 0
  local queue = sem.blocked          -- fila de espera
  if table.getn(queue) > 0 then
    local c = table.remove(queue,1) -- remove tarefa da fila
    table.insert(tasks,c)           -- e a reinsere na lista
  end                               -- de tarefas vivas
end
```

Figura A.3: Implementação de um semáforo binário

ou mesmo situações de *starvation*, se quando um semáforo é liberado, a co-rotina removida da fila de espera for colocada no final da lista de tarefas vivas. Soluções que tentem evitar esse tipo de problema podem ser desenvolvidas, por exemplo, colocando-se a co-rotina no início da lista de tarefas vivas, e/ou suspendendo a co-rotina que libera um semáforo, colocando-a no final da lista de tarefas. Entretanto, a necessidade de mecanismos de sincronização em aplicações que utilizam um modelo de gerência cooperativa é significativamente menor do que em ambientes que envolvem preempção. Para a maioria das aplicações, o cuidado em evitar situações como a descrita é desnecessário.

Semáforos contadores são tipicamente utilizados quando é necessário limitar o número de tarefas que compartilham, simultaneamente, um dado recurso. A implementação de um semáforo contador é bastante semelhante à de um semáforo binário. As modificações necessárias são a atribuição de um valor inicial ao semáforo (correspondente ao número máximo de tarefas que podem obtê-lo), o decremento desse valor cada vez que uma tarefa obtém o semáforo, o bloqueio de tarefas quando esse valor é 0, e o incremento do valor de um semáforo quando uma tarefa o libera.

Mecanismos de sincronização por condições também têm uma implementação trivial, apresentada na Figura A.4. Para representar uma condição, é suficiente uma tabela que representa a fila de tarefas bloqueadas. A espera por uma condição pode ser implementada por uma função

```
-- cria uma condição
function create_cond() return {blocked = {}} end

-- aguarda uma condição
function wait(cond)
  coroutine.yield("blocked", cond)
end

-- sinalização da condição (libera uma tarefa apenas)
function signal(cond)
  local queue = cond.blocked
  if table.getn(queue) > 0 then
    local c = table.remove(queue,1)
    table.insert(tasks,c)
  end
end

-- sinalização da condição (libera todas as tarefas)
function signal_all(cond)
  local queue = cond.blocked
  for i = 1, table.getn(queue) do
    table.insert(tasks, queue[i])
  end
  cond.blocked = {}
end
```

Figura A.4: Implementação de sincronização por condições

que suspende a co-rotina ativa, retornando ao escalonador um valor que indica que a co-rotina deve ser inserida na fila da condição especificada, e removida da lista de tarefas (ou seja, o mesmo procedimento implementado para aguardar um semáforo). As operações de sinalização removem uma, ou todas, as co-rotinas bloqueadas da fila da condição correspondente, reinserindo-a(s) na lista de tarefas. Na ausência de preempção, mecanismos de exclusão mútua, utilizados em implementações tradicionais de monitores em ambientes de *multithreading*, não são necessários.

A Figura A.5 apresenta a versão final do escalonador do nosso ambiente de gerência cooperativa de tarefas. Essa versão incorpora o suporte a operações de entrada e saída não bloqueantes, o uso de um *pool* de co-rotinas e o bloqueio de tarefas à espera de uma condição ou de um semáforo.

```
function dispatcher()
  while true do
    local n = table.getn(tasks)
    if n == 0 then break end
    resources = {}                -- tabela de recursos
    for i = 1, n do
      task = tasks[i]
      local res,s = task()        -- reativa tarefa
      if res == "task ended" then
        table.remove(tasks, i)    -- tarefa terminou
        break
      elseif res == "blocked" then
        table.remove(tasks, i)    -- tarefa bloqueada
        table.insert(s.blocked, task)
        break
      elseif res ~= nil then      -- tarefa aguarda E/S
        table.insert(resources, res)
      end
    end

    -- todas as tarefas vivas aguardam E/S ?
    if table.getn(resources) == table.getn(tasks) then
      select(resources)
    end
  end
end
```

Figura A.5: Gerência cooperativa de tarefas com sincronização