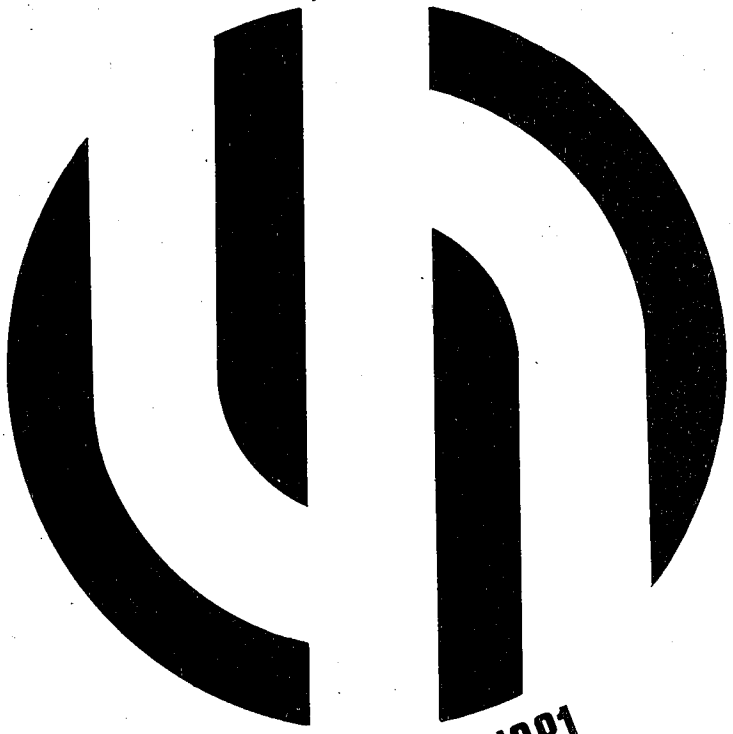
anais do 8º seminário integrado de SOFTWARE E HARDWARE



27 A 31 de julho de 1981
ufsc · florianópolis

004.06
S471
1981

Sociedade Brasileira de Computação
CNPq / Conselho Nacional de Desenvolvimento Científico e Tecnológico
SEI / Secretaria Especial de Informática
CAPES / Coordenação de Aperfeiçoamento de Pessoal do Ensino Superior
UFSC / Universidade Federal de Santa Catarina
Governo do Estado de Santa Catarina

ANAIS DO

VIIIº SEMISH

SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE

Iº CONGRESSO DA

SOCIEDADE BRASILEIRA DE COMPUTAÇÃO

FLORIANÓPOLIS, SANTA CATARINA

27 A 31 DE JULHO DE 1981

PROGRAMAÇÃO CONCORRENTE NUMA LINGUAGEM DE ALTO NÍVEL - UMA IMPLEMENTAÇÃO

Yussef Farran L.
Depto. de Ingeniaria de Sistemas
Universidade de Concepción
Casilla 1186
Concepción
Chile

Michael Stanton
Depto. de Informática
PUC/RJ
Gávea
22453 - Rio de Janeiro
Brasil

SUMÁRIO

Descreve-se a implementação de um sistema de programação concorrente, baseada nos conceitos de processos e monitores. Um programa concorrente é escrito em BCPL, seguindo certas convenções, e executa no processador virtual proporcionada por um sistema operacional convencional. (IBM OS/370). As rotinas de suporte de BCPL foram acrescidas de um núcleo de multi-programação, do qual a maior parte foi feita em BCPL. O software foi desenvolvido para suportar aplicações concorrentes, tais como protocolos de comunicação. Também ele poderá ser útil no ensino de programação concorrente.

ABSTRACT

This paper describes the implementation of a concurrent programming facility based on the concepts of processes and monitors. A concurrent program is written in BCPL, following certain conventions, and executes on the virtual processor provided by a conventional operating system (IBM OS/370). The BCPL run time library was extended to include a multi-programming kernel, which was largely programmed in BCPL. This software was designed to support concurrent applications, such as communications protocols. It may also be useful in teaching concurrent programming.

1. INTRODUÇÃO

A aplicação do computador para resolver um problema de controle requer a criação de um modelo computacional do sistema a ser controlado. Sistemas concorrentes (ou paralelos), onde os componentes do sistema atuam assincronamente, apresentam uma complexidade maior do que sistemas síncronos (ou sequenciais). Neste caso, o modelo computacional do sistema precisa refletir este paralelismo, sendo organizado como um conjunto de atividades paralelas, e assíncronas chamadas processos. Cada processo executa num processador virtual, implementado em hardware e software.

A primeira aplicação de sistemas paralelos deu-se na criação de sistemas operacionais, programas que controlam o próprio hardware do computador. Qualquer sistema atendendo a dois ou mais usuários precisa adotar um sistema paralelo. O modelo mais comum é o de uma família de processos que se coordenam mutuamente através de operações de sincronização implementadas num núcleo. A cada usuário corresponde um processo, e frequentemente o usuário dispõe das mesmas facilidades usadas pelo sistema operacional para organizar sua aplicação como um conjunto de processos paralelos. Deve-se notar que estas facilidades foram escolhidas pelos projetistas do sistema operacional, e, portanto, não podem ser alteradas. Em geral, estas facilidades representam uma tecnologia antiquada (semáforos), quando lembramos os avanços ocorridos na compreensão de programação concorrente durante a última década. Para controlar acesso a variáveis compartilhadas hoje em dia prefere-se usar o conceito de monitor, um tipo abstrato de dado encapsulando as variáveis compartilhadas e sincronizando acesso a elas [1,5]. Uma aplicação concorrente baseada nos conceitos de processo e monitor deveria ser implementada numa linguagem apropriada, tal como *Pascal concorrente* [2], *Modula* [15], ou *Ada* [9]. Porém, as implementações ora conhecidas destas linguagens requerem o uso dedicado do processador. Não se contemplou por ora o uso destas linguagens dentro do ambiente proporcionado por um sistema operacional convencional, onde o conceito de monitor não é utilizado.

O ponto de partida deste trabalho foi a decisão de implementar uma aplicação concorrente usando um computador (*IBM 370*) com um sistema operacional convencional (*OS/MVT*). *OS/MVT* oferece facilidades de multiprogramação para o usuário. Portanto o custo destes é relativamente elevado e o usuário não tem controle muito preciso sobre o escalonamento dos seus processos. Como a aplicação deveria utilizar um número elevado de processos, abandonamos imediatamente a idéia de implementar seus processos e monitores usando as facilidades de multiprogramação do sistema operacional. A alternativa ao uso destas facilidades seria implementar um núcleo de multiprogramação que multiplexasse en-

tre os processos (nível 2) da aplicação o processador virtual, em que executa o processo (nível 1) do usuário do sistema operacional. Existem diversos precedentes para este modo de implementar uma aplicação concorrente no sistema *OS/370* : o sistema de entrada de jobs *HASP* [7], e vários sistemas interativos, tais como *APL360*, *ROSCOE*, *WYLBUR* e *GUTS*, não utilizam as facilidades de multiprogramação do sistema operacional. Em cada um destes casos, um número variável de atividades em paralelo é coordenado através de um núcleo próprio (diferente em cada caso). Em geral, a linguagem usada nestas implementações foi *Assembler*.

À luz destes precedentes, foi definido como primeiro objetivo a criação de um sistema de programação concorrente em linguagem de alto nível, baseada no conceito de processos e monitores, e que pudesse ser usado no ambiente do sistema operacional *OS/370*. No que segue descrevemos a escolha da linguagem, o projeto do núcleo, e o uso do sistema criada.

2. ESCOLHA DA LINGUAGEM

Para diminuir o trabalho de implementação procurou-se usar uma linguagem geralmente disponível, que pudesse ser adaptada ao projeto. Qualquer linguagem sequencial serve para definir processos sequenciais, embora seja desejável que todas as procedures sejam reentrantes para permitir ativação múltipla. Este requisito exclui linguagens que não suportam recursão, tais como *Fortran* ou *Cobol*. A implementação de monitores seria facilitada pelo uso de uma linguagem que permitisse a definição de tipos abstratos de dados, como *Símula 67* [3], *Alphard* [16] ou *CLU* [10]. Se nos restringirmos a linguagens convencionais, monitores serão implantados como procedures, que manterão como variáveis locais os dados protegidos pelo monitor. Além de serem locais ao procedure, e assim invisíveis de fora, é necessário que estas variáveis lembrem dos seus valores entre atuações sucessivas. Este requisito exclui *Pascal* e *Algol 68*, para as quais variáveis locais são reinicializadas a cada ativação da procedure. Linguagens que satisfazem este segundo requisito incluem *Algol 60* (own), *PL/I* (*STATIC*) e *BCPL* (*STATIC*), que implementam mais de uma classe de variável local. A necessária sincronização entre processos usuários do monitor deverá ser implementada através de rotinas de núcleo chamadas dentro do monitor. Para isto, precisa-se da facilidade de poder definir procedures externas, compiladas em separado, possivelmente numa outra linguagem. As implementações de *PL/I* e de *BCPL* [14] disponíveis satisfazem este terceiro critério.

Tanto *PL/I* como *BCPL* já foram usadas para implementar monitores : Nehmer [13] descreve o uso de *PL/Z* (semelhante a *PL/I*) num microprocessador *Zilog Z-80* e Lister [11] descreve o uso de *BCPL* num *DEC PDP-10*. Nossa escolha baseou-se nas dificuldades percebidas na realização do núcleo. As duas lingua-

gens oferecem as mesmas facilidades de controle e estruturação de dados, através de tipos em *PL/I* e manipulação de ponteiros em *BCPL*, uma linguagem sem tipos. Como toda interação com o sistema operacional (principalmente E/S) é feita através da chamada explícita de rotinas em *BCPL*, torna-se mais fácil no caso desta linguagem interceptar estas interações do que no caso de *PL/I*, onde operações de E/S são comandos da linguagem. Outra vantagem de *BCPL* é a maior simplicidade do seu ambiente de execução, onde as variáveis dinâmicas de uma procedure não são visíveis para suas procedures internas. Como consequência disto, somente o valor atual do ponteiro da pilha de registros de ativação poderá ser suficiente para descrever totalmente o processo. Isto tem consequências importantes para o núcleo. Por outro lado, *PL/I* permite acesso a variáveis dinâmicas (*AUTOMATIC*) não locais, o que faz bem mais complexo seu ambiente de execução. Existe ampla documentação sobre o sistema de execução de ambas estas linguagens [8,12]. Finalmente foi escolhida a linguagem *BCPL*, por sua maior simplicidade como linguagem e no sistema de execução. Em retrospecto, esta decisão realmente simplificou o trabalho da implementação que pôde ser feito sem alterar o sistema de execução.

3. O NÚCLEO CONCORRENTE

A estrutura adotada para o núcleo segue basicamente a descrição adotada por Holt [6], com algumas extensões a serem descritas. As funções principais do núcleo são de multiplexar entre os processos de *BCPL* o processador virtual, fornecido pelo sistema operacional, e de implementar monitores. O processador virtual fornecido pelo sistema operacional *OS/370* tem as seguintes características gerais: o conjunto de instruções de hardware está limitado às instruções não privilegiadas da *IBM 370*, acrescidas de chamadas ao supervisor (*SVC*'s), que ativam rotinas do sistema operacional que utilizam as instruções privilegiadas (E/S e uso do relógio).

Destacamos a facilidade de relógio, pela qual poderá ser solicitada uma interrupção do processador virtual depois de expirado um dado intervalo de tempo. Usamos esta facilidade para incorporar um relógio de tempo real no processador virtual. Sua função será de evitar monopólio do processador virtual por um processo. Seu uso neste sistema é facultativo.

Representação de processos e monitores. É conveniente ver o núcleo como a implementação dos tipos abstratos processo e monitor. Enquanto estiver suspenso, o processo é representado por seu descritor, que contém o estado do processador virtual, o conteúdo dos registradores e o estado de E/S. No *IBM 370* o estado do processador virtual corresponde à segunda parte do *Program Status*

Word (PSW), que inclui o contador de instruções. Como *BCPL* convencionalmente suporta a noção de um arquivo corrente de entrada, e outro de saída, o estado de E/S consiste de ponteiros para os descritores destes arquivos.

Um processo é definido como a execução de uma procedure em *BCPL*. Cria-se um ou mais processos por uma única chamada da procedure -:

PARTIDA_PROCESSOS (intervalo, número-de-processos, tabela-de-processos)

que também inicia sua execução concorrente. Para cada processo a ser criado, a tabela-de-processos define os parâmetros: *procedure-inicial, tamanho-da-pilha, prioridade-de-execução, parâmetro-inicial*, os quais definem as condições iniciais do processo e a memória necessária à sua execução. A chamada de *PARTIDA_PROCESSOS* aloca o descritor e memória para a pilha de execução para cada um dos processos criados. A memória para estas alocações, como para todas as outras feitas pelo núcleo, é obtida da pilha de execução do único processo inicial, que chama *PARTIDA_PROCESSOS*. Este processo inicial só continua sua execução depois de terminados todos os processos criados por ele.

O núcleo geralmente encadeia os decritores numa fila de processos prontos, ordenada de acordo com a prioridade-de-execução. Depois de inicializado o relógio de tempo real, um processo é selecionado e despachado. Ao ocorrer uma interrupção do relógio, o processo em execução volta à fila de processos prontos, e outro processo é despachado.

Suporte para monitores consiste de quatro entradas do núcleo :

ENTRA_MON

SAI_MON

WAIT (condição, prioridade-de-espera)

SIGNAL (condição)

ENTRA_MON e *SAI_MON* são rotinas usadas para garantir acesso exclusivo a um monitor, e são chamadas ao entrar e sair de um monitor. Como trabalhamos com um só processador virtual, exclusão mútua pode ser garantida inibindo interrupções do relógio. O descritor de processo contém um campo inicialmente zero, que é incrementado a cada *ENTRA_MON* e decrementado a cada *SAI_MON*. Somente é permitida uma interrupção quando este campo tiver o valor 0, ou seja, quando o processo não está dentro de um monitor. Deve-se observar que são permitidas chamadas aninhadas de monitores. Quando um processo é suspenso dentro de um monitor interno, é liberada exclusão mútua para este monitor, e também para todos os monitores de níveis superiores. Isto pressupõe que, antes de fazer uma chamada aninhada a outro monitor, seja restaurado o invariante do monitor. As rotinas *WAIT* e *SIGNAL* operam em variáveis de condição, que são representadas por filas de descritores de processos, uma fila para cada variável. A chamada de *WAIT* enfileira o processo em questão na fila indicada de a-

cordo com a *prioridade-de-espera*. Em seguida será despachado um processo na fila de prontos. A chamada de *SIGNAL* numa condição não vazia resulta na transferência para a cabeça da fila dos processos prontos do processo liberado. Porém continuamos imediatamente a execução do processo que executou *SIGNAL*. Nota-se que aqui divergimos da especificação de Hoare para a operação *SIGNAL* [5], e seguimos a posição de Wirth na sua descrição de *Modula* [15].

Troca de Processos. Execução concorrente dos processos é simulada pela multiplexação do único processador virtual entre os processo. Um novo processo é despachado nos seguintes casos:

(i) um processo executa a operação *WAIT*.

(ii) ocorre uma interrupção do relógio e existe um processo pronto para executar, de prioridade maior, para substituir aquele que foi interrompido.

A rotina que despacha um processo inclui uma pequena parte em *Assembler* que manipula os registradores do hardware. A implementação de *BCPL* reduz em muito o trabalho de trocar processos, no caso de execução do comando *WAIT*. Neste caso o valor do ponteiro da pilha do processo descreve totalmente o estado do processo.

O caso do processo ser interrompido pelo relógio é mais complexo, porque é necessário salvar todos os registradores do hardware inclusive o registrador de estado (*PSW*). Este trabalho requer interação com o sistema operacional, que já salvou este contexto antes de ativar a rotina do núcleo que trata da interrupção. Esta rotina, necessariamente feita em *Assembler*, restaura o ambiente *BCPL* e chama uma rotina de *BCPL* para processar a interrupção. No caso de determinar a troca do processo, é necessário atualizar a área onde o sistema operacional salvou o contexto do processo interrompido. Como esta área está protegida, foi incorporada no sistema operacional uma rotina com esta finalidade ("*user SVC*"). Esta rotina também é invocada na execução de *WAIT* quando for desejado despachar um processo previamente interrompido pelo relógio. Neste caso, o privilégio do sistema operacional é necessário por ser impossível carregar todos os registradores e a segunda parte da *PSW* do *IBM 370* sem recorrer a uma instrução privilegiada.

A rotina que foi acrescentada ao sistema operacional é pequena (272 *bytes*) e representa a única modificação necessária para suportar multi-programação do processador virtual. Ela não depende das características de *BCPL* e poderá ser usada para implementar outras linguagens concorrentes num sistema *OS/370*.

Voltando ao assunto das interrupções do relógio, existem situações nas quais não podemos permitir uma troca de processo, por ameaçar a integridade da nossa implementação ou até do sistema operacional. São estas :

(a) execução de um monitor

(b) execução de certas rotinas de suporte de execução de *BCPL*

(c) execução de rotinas do sistema operacional

(d) execução da rotina que trata da interrupção

Já mencionamos que caso (a) é resolvido associando a cada processo um campo que indica a diferença entre o número de chamadas de *ENTRA_MON* e o número de chamadas *SAI_MON*. Caso (b) cria problemas pelo fato de que o suporte de execução mantém dados internos sobre o estado de E/S, e para proteger sua integridade, devemos considerar as rotinas básicas de E/S como não interrompíveis. Isto é feito por chamadas de *ENTRA_MON* e *SAI_MON* antes e depois de chamadas destas rotinas, e implementado transparentemente para os programas em *BCPL* através de mudanças de endereços guardados no vetor global. (O efeito disto é de serializar todas as operações de E/S , mas isto não nos parece muito grave.) Caso (c) ameaça a integridade do sistema operacional, e a rotina acrescentada ao sistema operacional não permite trocar o processo neste caso. O caso (d) não ocorre, porque o relógio interrompe somente uma vez e precisa ser reinicializado após cada interrupção.

Na sua maior parte, o núcleo descrito aqui foi programado em *BCPL*, necessitando apenas de algumas rotinas pequenas (736 *bytes*) em *Assembler* usadas para manipulação de registradores de hardware, e para fazer interface com o sistema operacional (uso do relógio, e chamadas da rotina de despacho). Inclusive a rotina de tratamento de interrupções é feita em *BCPL*. Este uso de *BCPL* em muito facilitou o desenvolvimento do núcleo, e permite ainda a flexibilidade de poder mudar facilmente a implementação quando for desejado.

4. A PROGRAMAÇÃO DE MONITORES E PROCESSOS

Uma das maiores vantagens que advêm do uso de linguagens de alto nível é a definição precisa de interfaces através de regras de visibilidade [2,5,11]. Já foi discutido na seção 2 o uso de variáveis com o atributo *STATIC* para implementar as variáveis particulares de um monitor. Outras características de *BCPL* que facilitam a definição de interfaces são: (a) compilação em separado, e (b) a inclusão de texto num programa em tempo de compilação através da diretiva *GET*. Para facilitar a ligação de módulos compilados em separado, a diretiva *SECTION* define um nome para o módulo, e a diretiva *NEEDS* declara uma referência externa. O editor de ligações resolverá todas estas ligações.

O programador de uma aplicação concorrente deverá compilar seu programa em tres tempos: os monitores, os processos concorrentes e o processo inicial. O núcleo concorrente é ligado ao programa através da diretiva *NEEDS "NUCLEO"*, e a declaração da sua interface (*NUCHDR*) deverá ser incluida na compilação dos monitores e do processo inicial através da diretiva *GET "NUCHDR"*. O usuário

deverá definir a interface de suas monitores (*MONHDR*), e incorporá-la nas compilações através de *GET "MONHDR"*. A diretiva *GET "LIBHDR"* incorpora a interface com o sistema de execução de BCPL.

Como exemplo ilustrativo, mostramos a seguir a implementação parcial da gerência de um buffer circular, descrita na p.74 de [6]. Primeiro apresentamos o módulo *MONITOR*.

```
SECTION "MONITOR"
NEEDS   "NUCLEO"
GET     "LIBHDR"
GET     "NUCHDR"
GET     "MONHDR"
LET message_queue_monitor (entry,parm) BE
$(mqm
 MANIFEST $(m qb=5 // número de buffers a usar
          $)m
 STATIC $(stt
       first = NIL // ponteiro ao primeiro buffer ocupado
       tail  = NIL // ponteiro ao ultimo buffer ocupado
       q_full = 0  // quantidade de buffer ocupados
       buffer_vacant = CONDITION    // variáveis de
       buffer_occupied = CONDITION  // condição
       buffer = 0  // endereço inicial do buffer circular
       primeira_vez = TRUE // indicação para inicialização
       $)stt
LET spool_ (contents) BE
$(sp IF q_full = qb THEN WAIT (@buffer_vacant,0)
     buffer!tail := contents
     tail := (tail+1) REM qb
     q_full +:=1
     SIGNAL(@buffer_occupied)
$)sp
AND unspool_ (address_contents) BE
$(uns IF q_full = 0 THEN WAIT (@buffer_occupied,0)
     !address_contents := buffer!head
     head := (head+1) REM qb
     q_full -:= 1
     SIGNAL(@buffer_vacant)
$)uns
```

```
// entrada ao MONITOR
ENTRA_MON()
IF primeira_vez THEN $(pv
     buffer := GET_HEAP(qb) // aloca espaco para buffer
     primeira_vez := FALSE
     $)pv
SWITCHON entry INTO  // seleção da proc.
$(sw CASE spool   : spool_ (parm); ENDCASE
     CASE unspool : unspool_ (parm); ENDCASE
     DEFAULT      : print (erro_message); ENDCASE
$)sw
SAI_MON()
$)mqm
```

O arquivo *MONHDR* (segue) contém a declaração da interface do monitor.

```
MANIFEST   $(m     spool = 0;     unspool = 1   $)m
GLOBAL     $(g     message_queue_monitor : 200        $)g
```

Deve-se notar que as chamadas do monitor incluem como primeiro parâmetro um constante (*MANIFEST*) que seleciona a entrada desejada. A chamada é feita desta forma porque em *BCPL* as rotinas internas de uma procedure são invisíveis de fora dela. A inicialização das variáveis particulares do monitor ocorre na primeira chamada. As variáveis de condição são inicializadas com o valor *CONDITION* (definido em *NUCHDR*).

Em seguida apresentamos o módulo *PROCESSO*.

```
SECTION "PROCESSO"
NEEDS   "MONITOR"
GET     "LIBHDR"
GET     "MONHDR"
GET     "PROCHDR"
LET producer_process ()_ BE
$(pdp LET char = ?
     $(w char := RDCH()
          message_queue_monitor (spool, char)
     $)w   REPEATUNTIL char = ENDSTREAMCH
$)pdp
```

```
LET consumer_process ()_ BE
$(cnp LET char = ¬ENDSTREAMCH
     WHILE char ¬= ENDSTREAMCH DO
     $(w message_queue_monitor(unspool, @char)
        WRCH(char)
     $)w
$)cnp
```

O arquivo *PROCHDR* contém as identificações globais dos processos.

```
GLOBAL $(g  producer_process : 201;  consumer_process : 202  $)g
```

Para completar o programa, falta somente o processo inicial. Este consiste de uma procedure com o nome *START*, que inicia a ativação dos processos concorrentes com uma chamada da primitiva *PARTIDA_PROCESSOS* (v. seção 3). Uma especificação de *START* poderá ser encontrada em [4].

5. CONCLUSÕES

O sistema descrito neste artigo foi desenvolvido para facilitar a implementação do protocolo *X-25* usando uma tecnologia moderna de programação concorrente. Esta aplicação já foi parcialmente realizada [4], e assim sentimos que o objetivo final foi plenamente atingido. Estamos confiantes que futuras aplicações concorrentes também poderão utilizar o software enquanto não aparecer um melhor. Além desta utilidade, uma outra aplicação já prevista para "*BCPL concorrente*" é no ensino de programação concorrente. Como mencionados antes, as rotinas que foram incorporadas ao sistema operacional são de propósito geral, e antecipamos sua utilização nas implementações de outras linguagens concorrentes em *OS/370*.

6. AGRADECIMENTOS

Os autores agradecem apoio financeiro da Finep e do CNPq, ambos do Brasil. -Um dos autores (Y.F.L.) recebeu apoio também de sua universidade durante sua estadia no Brasil. O computador usado foi o *IBM 370/165* da Pontifícia Universidade Católica do Rio de Janeiro.

Uma versão mais extensa deste trabalho será apresentada na Primeira Conferência Internacional em Ciência da Computação, Santiago, Chile, em agosto de 1981.

7. REFERÊNCIAS

[1] Brinch Hansen,P., "Structured Multiprogramming", *Communications of the* ACM, vol 15, no. 7, pp 574-578, julho de 1972.

[2] Brinch Hansen,P., *The Architecture of Concurrent Programs*, Prentice Hall, Englewood Cliffs, EE.UU., 1977.

[3] Dahl,O.J., Myhrhang,B., Nygaard,K., "Simula 67 - Common base Language", Norge Regnesentral, Oslo, Noruega, 1968.

[4] Farran L.,Y.E., "Especificação de uma implementação da Recomendação X-25 do C.C.I.T.T. para um sistema IBM/370", tese de mestrado, Depto. de Informática, PUC, Rio de Janeiro, 1981.

[5] Hoare,C.A.R., "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, vol 17, no. 10, pp 549-557, outubro de 1974.

[6] Holt,R.C., Graham,G.S., Lazowska,E.D., Scott,M.A., *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley, Reading, EE.UU..,1978.

[7] IBM Corporation, *The HASP System*, 1971.

[8] IBM Corporation, *PL/I (F) Subroutine Library Program Logic*, Form number GY28-6801 - 6a. edição, 1972.

[9] Ichbiah,J. e outros, "Reference Manual for the Ada programming language", *Sigplan Notices*, vol.14, no. 6, Parte A, junho de 1979.

[10] Liskov,B.H., Syndler,A., Atkinson,R., Schaffert,C., "Abstraction Mechanisms in CLU", *Sigplan Notices*, vol 10, no. 6, pp 534-545, junho de 1975.

[11] Lister,A.M., Maynard,K.J., "An Implementation of Monitors", *Software - Practice and Experience*, vol 6, pp 377-385, 1976.

[12] Moody,K., "The BCPL System for IBM 360/370 Computers", Kings College, Cambridge, England, 1978.

[13] Nehmer,J., "The Implementation of Concurrency for a PL/I-like language", *Software - Practice and Experience*, vol 9, pp 1043-1057, 1979.

[14] Richards,M., Whitby-Stevens,C., *BCPL: The Language and its Compiler*, Cambridge University Press, 1980.

[15] Wirth,N., "Design and Implementation of Modula", *Software - Practice and Experience*, vol 7, pp 67-84, 1977.

[16] Wulf,W.A., London,R.L., Shaw,M., "Abstraction and Verification in Alphard", *IEEE Transactions on Software Engineering*, abril de 1976.