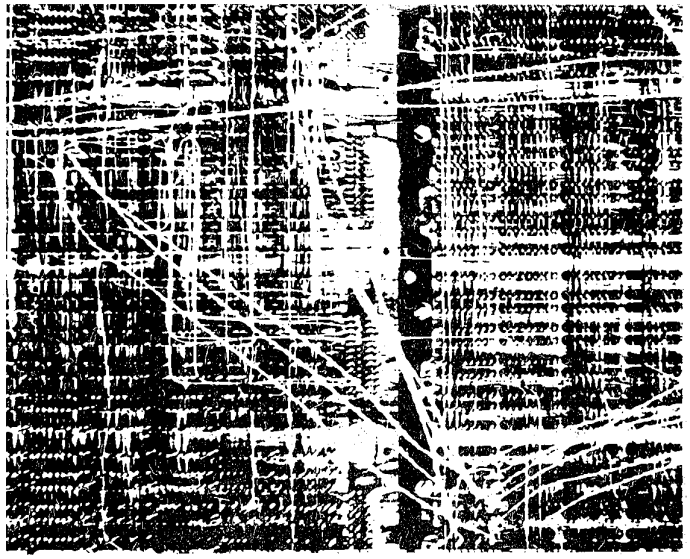

ACTAS DE LA **PRIMERA CONFERENCIA INTERNACIONAL EN CIENCIA DE LA COMPUTACION**

Auspician:
Banco de Chile •
Secico •
A.F.P. Santa María •
Leniz y Silva, •
Ingenieros Consultores
Cientec •
Cecinas Winter •
Revista Informativa •

PONTIFICIA UNIVERSIDAD
CATOLICA DE CHILE

UNIVERSIDAD DE CHILE

PRIMERA CONFERENCIA INTERNACIONAL

EN CIENCIA DE LA COMPUTACION

FECHA : 24 - 27 AGOSTO

LUGAR : CASA CENTRAL
PONTIFICIA UNIVERSIDAD
CATOLICA DE CHILE

PROGRAMACIÓN CONCURRENTE USANDO MONITORES
- UNA IMPLEMENTACIÓN -

PROGRAMAÇÃO CONCORRENTE USANDO MONITORES
- UMA IMPLEMENTAÇÃO -

CONCURRENT PROGRAMMING USING MONITORS
- AN IMPLEMENTATION -

- Yussef Farran L.
  Departamento de Ingeniería de Sistemas
  Universidad de Concepción
  Chile

- Michael Stanton
  Departamento de Informática
  PUC - Rio de Janeiro
  Brasil

Resumo

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 runtime 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 [Brinch Hansen 72, Hoare 74]. Uma aplicação concorrente baseada nos conceitos de processo e monitor deveria ser implementada numa linguagem apropriada, tal como Pascal concorrente [Brinch Hansen 77], Modula [Wirth 77], ou Ada [Ichbiah 79]. 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 entre os processos (nível 2) da aplicação o processador

[Brinch Hansen 77] P.Brinch Hansen; The Architecture of Concurrent Programs. Prentice Hall, 1977.

[Dahl 68] O.J.Dahl, B.Myhrhang e K.Nygaard; "Simula 67 - Common base Language", Norge Regnesentral, Oslo, Noruega.

[Farran 81] Y.E.Farran L.; "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/RJ.

[Hoare 74] C.A.R.Hoare; "Monitors: An Operating System Structuring Concept". Communications of the ACM, vol 17, no. 10, Outubro 1974, pp 549-557.

[Holt 78] R.C.Holt, G.S.Graham, E.D.Lazowska and M.A.-Scott; Structured Concurrent Programming with Operating Systems Applications. Addison-Wesley, 1978

[IBM 71] The HASP System. 1971.

[IBM 72] IBM PL/I Subroutine Library Program Logic; Form number GY28-6801 - 6a. edição.

[Ichbiah 79] J. Ichbiah e outros; "Reference Manual for the Ada programming language"; Sigplan Notices, vol 14 (6), Parte A.

[Liskov 75] B.H.Liskov, A.Syndler, R.Atkinson e C.Schaffert; "Abstraction Mechanisms in CLU"; Sigplan Notices, vol 10 (6), pp 534-545.

[Lister 76] A.M.Lister and K.J.Maynard; "An Implementation of Monitors". Software - Practice and Experience, vol 6, 1976, pp 377-385.

[Moody 78] K.Moody; The BCPL System for IBM 360/370 Computers. Kings College, Cambridge, England.

[Nehmer 79] J.Nehmer; "The Implementation of Concurrency for a PL/I-like language". Software - Practice and Experience, vol 9, pp 1043-1057

[Richards 79] M.Richards and C.Whitby-Stevens; BCPL: The Language and its Compiler. Cambridge University Press, 1980.

[Wirth 77] N.Wirth; "Design and Implementation of Modula", Software - Practice and Experience, vol 7, pp 67 a 84.

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 [IBM 71], e vários sistemas interativos, tais como APL360, ROSCOE, WYLBUR e GUTS, não utilizam os sistemas 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 Simula 67 [Dahl 68], Alphard [Wulf 76] ou CLU [Liskov 75]. 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 [Richards 79] disponíveis satisfazem este terceiro critério.

Tanto PL/I como BCPL já foram usadas para implementar monitores : Nehmer [Nehmer 79] descreve o uso de PL/Z (semelhante a PL/I) num microprocessador Zilog Z-80 e Lister [Lister 76] descreve o uso de BCPL num DEC PDP-10. Nossa escolha baseou-se nas dificuldades percebidas na realização do núcleo. As duas linguagem oferecem as mesmas faci-

lidades 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 são invisíveis fora dela. 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. [Moody 78, IBM 72]. 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 [Holt 78], 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. O uso deste 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 de 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 susperiores. 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 acordo com a prioridade-de-espera. Em seguida será despachado um processo na fila de pontos. A chamada de SIGNAL numa condição não

vazia resulta na transferência para a cabeça da fila dos processos prontos o 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 [Hoare 74], e seguimos a posição de Wirth na sua descrição de Modula [Wirth 77].

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 quando existir um processo pronto para executar de prioridade 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.

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 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 a 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. O Uso do Sistema de Programação Concorrente

As características de BCPL e suas regras de escopo permitem verificar certos requisitos a cumprir pela implementação do conceito de monitor em tempo de compilação [Hoare 74, Lister 76]. A compilação em separado dos módulos que constituem o SPC (Sistema de Programação Concorrente), permitirá que os monitores só vejam o que o núcleo quer mostrar, ou seja, as primitivas oferecidas, e além delas, o resto do núcleo será "invisível". Usando essa técnica de compilação em separado, é conveniente que os processos também sejam compilados em separado dos monitores, para garantir na compilação dos módulos que os procedimentos que implementam os processos estejam isolados dos monitores, conseguindo ter acesso a eles só através de suas entradas assim impedindo o acesso direto aos dados protegidos.

O implementador de monitores usará no cabeçalho do módulo os comandos NEEDS "NÚCLEO" para a 'linkage-edição' e GET "NUCMON" para a compilação. (1) NUCMON é um arquivo provido pelo núcleo para a interface com os monitores, mostrando através dele os números globais usados pelas primitivas e variáveis comuns.

Cada monitor é implementado como uma rotina BCPL e seus procedimentos como rotinas aninhadas dentro dele. Isto causa que estes procedimentos fiquem inacessíveis de fora do monitor. Para solucionar este problema, os processos que solicitem o uso de um procedimento dentro de um monitor, farão a chamada ao monitor colocando como parâmetro o nome do procedimento desejado [Lister76].

As variáveis do MONITOR, de condição e outras, são declaradas na cláusula STATIC, sistema provido por BCPL que permite que estas variáveis sejam inicializadas só uma vez e não cada vez que o monitor é entrado (chamado).

Para tentar explicar melhor o SPC e seu uso, se mostrará a implementação do exemplo de gerência de um buffer circular descrito no [Holt78] pp 74.

O módulo MONITOR é escrito como segue:

```
001 SECTION "MONITOR"
002 NEEDS   "NUCLEO"
003 GET     "LIBHDR"
004 GET     "NUCMON"
005 GET     "USERLIB"
006
007 LET message_queue_monitor (entry,parm) BE
008 $(mqm
009  MANIFEST $(m qb=5 // quantidade de buffers a usar
010             $)m
011  STATIC $(stt
012       first = NIL // ponteiro ao primeiro buffer ocupado
013       tail  = NIL // ponteiro ao ultimo buffer ocupado
014       q_full = 0  // quantidade de buffer ocupados
015
016       buffer_vacant = CONDITION    // uma condição
017       buffer_occupied = CONDITION  // outra condição
018
019       buffer = 0  // endereço inicial do buffer circular
020       primeira_vez = TRUE // indicação para inicializa-
021       $)stt               // ção do monitor
022
023 LET spool_ (contents) BE
024 $(sp IF q_full = qb
```

---

(1) NEEDS é uma diretiva de BCPL que avisa do uso de rotinas compiladas em separado. GET é uma diretiva que permite incluir no texto sendo compilado o conteúdo de um arquivo externo. GET normalmente é usado para incluir declarações comuns a módulos compilados em separado; convencionalmente GET "LIBHDR" incorpora a definição das rotinas do sistema de execução ("runtime library").

```
025             THEN WAIT (@buffer_vacant,0)
026       buffer!tail := contents
027       tail := (tail+1) REM qb
028       q_full +:=1
029       SIGNAL(@buffer_occupied)
030       RETURN
031 $)sp
032
033 AND unspool_ (address_contents) BE
034 $(uns IF q_full = 0
035             THEN WAIT (@buffer_occupied,0)
036       !address_contents := buffer!head
037       head := (head+1) REM qb
038       q_full -:= 1
039       SIGNAL(@buffer_vacant)
040       RETURN
041 $)uns
042
043 // entrada ao MONITOR
044 ENTRA_MON()
045 IF primeira_vez THEN $(pv
046       buffer := GET_HEAP(qb) // aloca espaco para buffer
047       primeira_vez := FALSE
048       $)pv
049 SWITCHON entry INTO  // seleção da proc.
050 $(sw CASE spool   : spool_ (parm); ENDCASE
051       CASE unspool : unspool_ (parm); ENDCASE
052       DEFAULT      : print (erro_message); ENDCASE
053 $)sw
054 SAI_MON()
055 $)mqm
```

As variáveis de condição são declaradas em STATIC atribuindo a elas o valor inicial CONDITION (palavra declarada em NUCMON), permitindo assim ao núcleo reconhecer e criar a fila de condição correspondente.

O comando IF da linha 45 faz com que o monitor seja inicializado, ou seja, a lista de comandos incluidos no bloco do THEN será executada só uma vez e nela o monitor alocará e inicializará os recursos necessitados.

O comando SWITCHON (linha 49) permite selecionar o procedimento solicitado pelo processo que está entrando no monitor, indicando através do parâmetro 'entry' o nome da rotina desejada. Pode-se notar que o parâmetro é um valor constante que permite realizar a seleção, para o qual será necessário fazer distinção entre o nome dado ao parâmetro e o nome dado ao procedimento, tentando manter a semelhança. Lister [Lister76] sugere usar o sistema de BCPL de permitir o "under-score" como componente de um nome BCPL, então, se usaram nomes iguais adicionando um "under-score" no nome do procedimento no monitor.

Os valores constantes usados como parâmetros serão declarados num cabeçalho comum para o módulo do monitor e

módulo do processo, através de uma cláusula MANIFEST. No exemplo o cabeçalho estará no arquivo "USERLIB" o qual é usado na compilação de monitores e processos.

O módulo de processos que usarão este monitor é o seguinte:

```
001 SECTION "PROCESSO"
002 NEEDS   "MONITOR"
003 GET     "LIBHDR"
004 GET     "USERLIB"
005
006 LET producer_process () BE
007 $(pdp LET char = ~ENDSTREAMCH
008       WHILE char ~= ENDSTREAMCH DO
009       $(w char := RDCH()
010           message_queue_monitor (spool, char)
011       $)w
012 $)pdp
013
014 LET consumer_process () BE
015 $(cnp LET line = ~ENDSTREAMCH
016       WHILE line ~= ENDSTREAMCH DO
017       $(w message_queue_monitor(unspool, @line)
018           WRCH(line); NEWLINE()
019       $)w
020 $)cnp
```

No intercâmbio de parâmetros entre rotinas e subrotinas deve-se lembrar que BCPL só passa parâmetros "por valor", mas que pode-se simular efetivamente a passagem de parâmetros "por referência" entregando como parâmetro o endereço da variável onde quer-se armazenar o resultado entregue pela sub-rotina. Note-se este caso no processo "consumer_process" linha 17 e no monitor linha 36.

O arquivo "USERLIB" é mostrado a seguir e contém:

a) a cláusula MANIFEST para declaração das constantes que identificam as entradas ao monitor. Se usa um "bit-pattern" relativamente complexo para reduzir a probabilidade de um acesso ao monitor com parâmetro 'entry' "acidentalmente" errado e constante certa.

b) a cláusula GLOBAL com os nomes dos procedimentos que implementam os processos.

```
MANIFEST $(m
     spool = 's'<<24 | 'p'<<16 | 'o'<<8 | 'l'
   unspool = 'u'<<24 | 'n'<<16 | 's'<<8 | 'p'
  $)m

GLOBAL $(g
    producer_process : 200
    consumer_process : 201
  $)g
```

Pela própria estrutura de BCPL, requer-se a existência da rotina START(), a qual é chamada pelo sistema BCPL para iniciar a execução. No SPC é usada para que o usuário descreva os processos a concorrer e para inicializar o SPC. Para o caso do exemplo aqui desenvolvido, a rotina START() é a seguinte:

```
001 NEEDS "PROCESSO"
002 GET   "LIBHDR"
003 GET   "USERLIB"
004 GET   "NUCMON"
005
006 LET START () BE
007 $(stt
008    MANIFEST $(
009       qp = 2       // quantidade de processos a concorrer
010       time_slice = 10  // fatia de tempo para cada proc.
011      $)

012    LET tabela = VEC (qp*TTG - 1) // cria tabela para
013                                  // descrição de processos
014    LET n,proc = 0,?          // variáveis auxiliares
015    USA_PRIOR := TRUE         // inicializa variável global
016                              // do núcleo.
017    // primeiro processo na tabela
018    n +:= 1
019    proc:= n*TTG-TTG + tabela // ptr. a entrada na tabela
020    PROC_ADDR OF proc :=  producer_process
021    TMAX_STACK OF proc := 100    // quantidade de palavras
022                        // precisadas para a pilha do processo
023    PRIORITY OF proc := 3  // prioridade relativa de exec.
024    PARM OF proc := 5
025
026    // segundo processo
027    n +:= 1
028    proc:= n*TTG-TTG + tabela // ptr. a entrada na tabela
029    PROC_ADDR OF proc := consumer_process
030    TMAX_STACK OF proc := 100
031    PRIORITY OF proc := 2 // prioridade relativa de exec.
032
033
034    PARTIDA_PROCESSOS(time_slice, qp, tabela) // start SPC
035 $)STT
```

A rotina START() tem um esquema relativamente rígido. Nela é criada a tabela de geração (linha 12) da figura 4.3, de tamanho dependente do número de processos 'qp' a descrever e do tamanho de cada entrada na tabela (TTC), inicializa a variável global USA_PRIOR, condição para enfileiramento com ou sem prioridade dos descritores na fila dos 'prontos', e descreve cada processo dando o "entry point" dele (linhas 20 e 29), o espaço precisado para o stack do processo (linha 21 e 30), a prioridade de execução relativa do processo (linhas 23 e 31) e se precisar, também pode-se entregar um parâmetro através de 'PARM OF proc' (linha 24).

Se a variável USA_PRIOR for inicializada com valor FALSE não é necessário incluir o valor da PRIORITY. As prioridades relativas de execução variam entre 0 (a maior) e 15 (a menor) sendo diferentes das prioridades de enfileiramento de descritores usadas pelo núcleo.

O nome do processo, que indica o endereço do início do código do processo, deverá estar no vetor global, definido no arquivo "USERLIB" para permitir a START() conseguir o valor correspondente, quando a compilação de START e PROCESSOS for em separado.

No procedimento START(), deverá existir pelo menos uma vez a primitiva PARTIDA_PROCESSOS para começar as atividades do SPC, que no exemplo atribui 10 centésimos de segundo para a fatia de tempo de cada processo e ativa 2 processos.

No entanto, quando todos os processos ativados terminarem sua execução, o NÚCLEO voltará o controle para o programa START() que reassumirá a execução, conseguindo se desejado, ativar um outro grupo de processos.

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 [Farran 81], 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 paralelo. 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.

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.

Referências Bibliográficas -

[Brinch Hansen 72] P.Brinch Hansen; "Structured Multiprogramming". Communication of the ACM, vol 15, no. 7, Julho 1972, pp 574-578.