Um Template Destinado à Construção de Sistemas Operacionais Para o VXt

Mauro M. Mattos, Antonio C. Tavares[1], Jorge S.Farias[2],Daniel S. Estrázulas[3]

Universidade Regional de Blumenau (FURB)
CEP – 89035-160 – Blumenau – SC – Brasil

União Metropolitana de Educação (UNIME)
CEP - 42700-000 - Lauro de Freitas - BA - Brasil

[1,3]{mattos,Tavares,estrazulas}@inf.furb.br, [2]jorgefarias@unime.com.br

Resumo. *Este artigo descreve o estágio atual do desenvolvimento de uma ferramenta de apoio ao ensino de conceitos básicos de sistemas operacionais e arquitetura de computadores. O VXt – virtual XT é um simulador de uma máquina baseada no processador Intel 8086. O presente trabalho descreve a estrutura de um template que serve como base para a construção de sistemas operacionais para o ambiente didático VXt. É descrita a arquitetura do hardware simulado bem como a funcionalidade do template.*

1 Introdução

Conforme Gagné, Brigss e Wagner (1992 apud CASAS, 1999), instrução, no contexto da escola, é um processo oferecido por instrutores humanos e envolve o professor e o aprendiz. Os méritos da instrução proporcionada por professores humanos são os aspectos dinâmicos e interativos do ensino.

Segundo Fagundes (1998) apud Veiga (2001), aprender por projetos é uma forma inovadora de romper com as tradições educacionais, dando um formato mais ágil e participativo ao trabalho de professores e educadores. Trata-se mais do que uma estratégia fundamental de aprendizagem, sendo um modo de ver o ser humano construir, aprendendo pela experimentação ativa do mundo. Ao elaborar seus projetos, o professor conduzirá seus alunos a um conjunto de interrogações, quer sobre si mesmos, quer sobre o mundo à sua volta, levando o aluno a interagir com o desconhecido ou com novas situações, buscando soluções para os problemas.

Conforme Maziero (2002), "uma das principais características da disciplina de Sistemas Operacionais é a relativa dificuldade em definir um seqüenciamento didático claro entre seus diferentes tópicos".

Conforme Machado e Maia (2004), "diversos pesquisadores como Ramakrishman e Lancaster (1993), Pérez-Dávila (1995), Fekete e Greening (1996), Wagner e Ressler (1997) e Downey (1999) propõem "*closed labs*", utilizando sistemas reais, na maioria dos casos, alguma versão do sistema operacional Unix".

Neste contexto, o presente trabalho descreve o estágio atual do projeto VXt – Virtual XT. O projeto, de caráter didático, consiste na implementação em software de uma máquina baseada no processador 8086 e vem sendo desenvolvido através de atividades de sala de aula, trabalhos de conclusão de curso

(LINZMEIER,1999;GOEDERT,2006) e através de projetos de iniciação científica (MATTOSet al.,2004). A versão atual caracteriza-se pela facilidade de propagação da interface com o usuário para uma classe de tal forma a permitir o acompanhamento passo a passo da execução de um programa. O software vem sendo utilizado em atividades de apoio ao ensino de sistemas operacionais e tem se mostrado uma ferramenta bastante útil.

A próxima seção descreve as principais características. A seção 3 descreve a arquitetura do sistema e as técnicas utilizadas. Na seção 4 são apresentados os resultados e discussões.

2 Arquitetura do sistema

A arquitetura do VXt caracteriza-se por adotar uma solução híbrida envolvendo componentes escritos em C, Delphi e Java. A justificativa para este modelo de desenvolvimento é apresentada a seguir.

O VXt, desde a sua primeira versão, foi implementado em Delphi. Em 2003, substituiu-se a implementação do conjunto de instruções que até então era implementado em Delphi pela biblioteca SoftX86 (implementada em C na forma de DLL). Esta decisão levou em consideração as principais características da biblioteca (SOURCEFORGE, 2003): (i) precisão na emulação da CPU; (ii) tamanho da memória ajustável; (iii) projeto que modela o comportamento dos seguintes sinais de hardware (na forma de *callbacks* no programa hospedeiro): *clock*, acesso ao espaço de E/S, acesso ao espaço de endereçamento da memória, interrupções por software, interrupções de hardware. Segundo (SOURCEFORGE, 2003) a biblioteca Softx86 foi concebida de forma a viabilizar a construção de programas de emulação de hardware. A biblioteca emula somente a CPU (figura 2). A simulação dos demais componentes de hardware deve ser realizada por um programa externo – neste caso, o VXt é responsável pela construção de toda a infra-estrutura adicional (*motherboard*, memória, espaço de endereçamento de Entrada/Saída, periféricos, etc.).

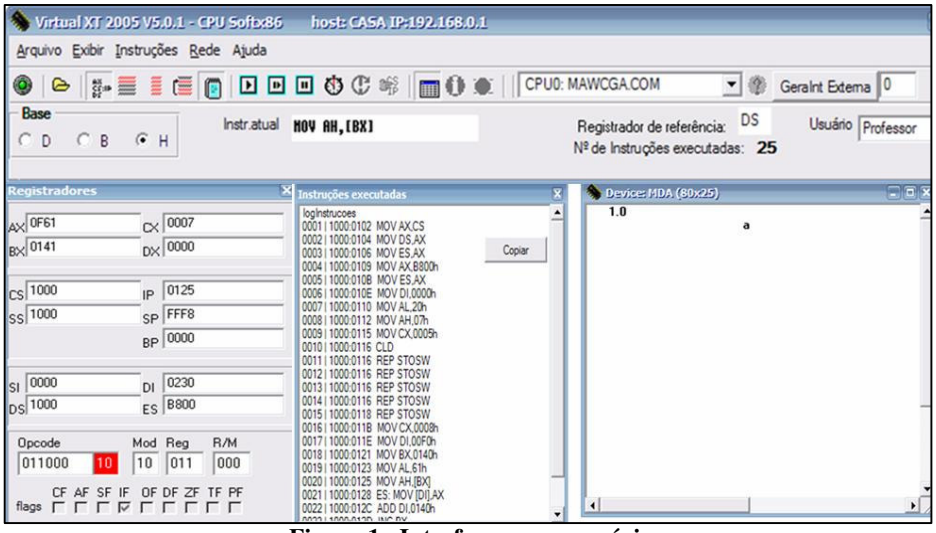

Figura 1. Interface com o usuário

O sistema foi concebido segundo o padrão *model-view-controller* (MVC) o qual oferece algumas características que fazem dele uma boa escolha de padrão a ser utilizado no desenvolvimento de software, dentre as quais, tem-se (SILVEIRA, 2006, p. 29):

a) Separar dados (*Model*) da interface do usuário (*View*) e do fluxo da aplicação (*Controller*);
b) Permitir que uma mesma lógica de negócio possa ser acessada e visualizada através de várias interfaces;
c) A lógica de negócio é independente da camada de interface com o usuário (*View*).
d) No contexto do projeto VXt, o padrão MVC foi adotado na seguinte perspectiva:
e) A versão em Delphi implementa o modelo do emulador do PC-XT e boa parte do modelo do controlador (uma vez que, o controle da aplicação ainda permanece na aplicação Delphi);
f) O *middleware* implementa parte do controlador da aplicação (uma vez que é o módulo responsável pela propagação das informações para os clientes e também por propagar os comandos de abertura/fechamento de janelas);
g) O módulo de interface do aluno implementa a camada de visão.

3 O middleware

Segundo Carvalho (2005), *middleware* é o neologismo criado para designar camadas de software que não constituem diretamente aplicações, mas que facilitam o uso de ambientes ricos em tecnologia da informação. A camada de *middleware* concentra serviços como identificação, autenticação, autorização, diretórios e outras ferramentas para segurança. Aplicações tradicionais implementam vários destes serviços, tratados de forma independente por cada uma delas. As aplicações modernas, no entanto, delegam e centralizam estes serviços na camada de *middleware*. Ou seja, o *middleware* serve como elemento que aglutina e dá coerência a um conjunto de aplicações e ambientes (CARVALHO, 2005).

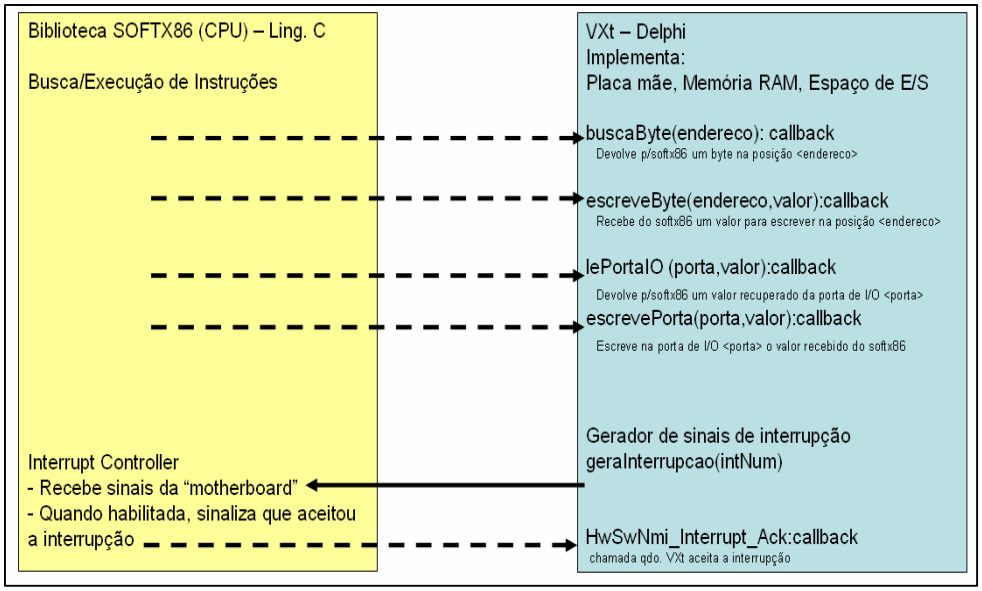

Figura 2 – Arquitetura do VXt utilizando a biblioteca Softx86.

Internamente, o *middleware* está estruturado na forma de um clássico modelo produtor/consumidor (CARISSIMI; OLIVEIRA; TOSCANI, 2003, p. 93). Ao receber uma mensagem do VXt, o servidor insere-a em um *buffer* global. Para cada cliente conectado é criada uma instância de um consumidor (*thread*) o qual, por sua vez, comunica-se exclusivamente com o seu cliente VXt Java. A figura 3 apresenta um diagrama de classes envolvendo os diversos componentes da camada *middleware*.

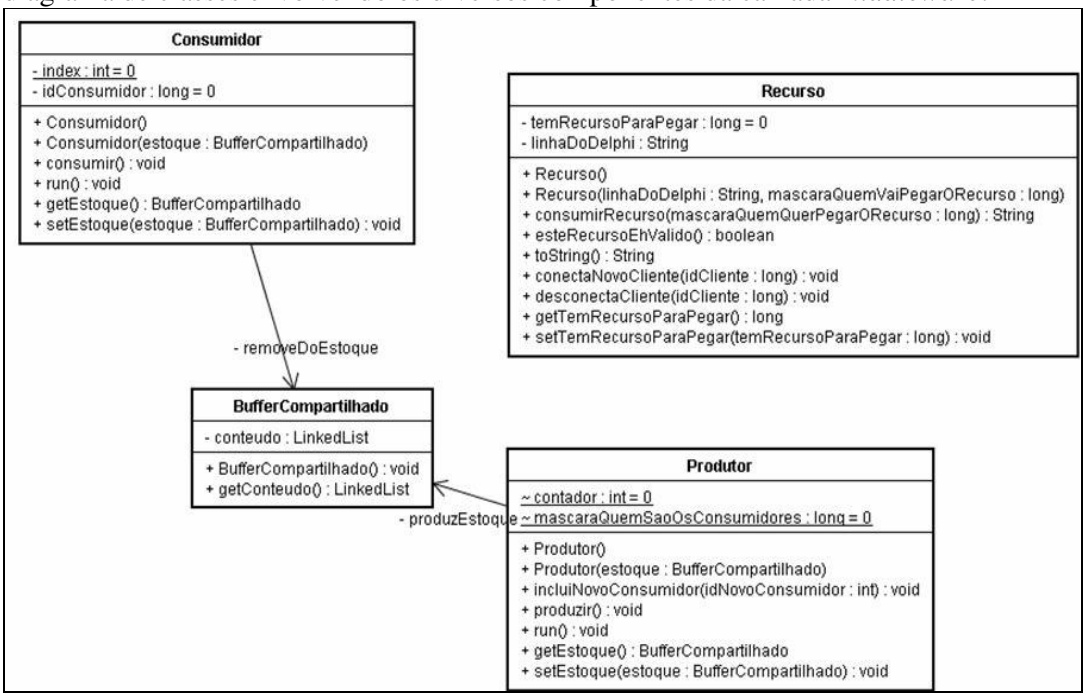

Figura 3 - Diagrama de classes do módulo middleware

A estratégia adotada para a concepção da arquitetura do *middleware* é apresentada na figura 4. Como se pode observar, o *middleware* atua como um cliente que requisita informações que o VXt em Delphi (atuando como servidor) fornece.

A estratégia baseada em produtor/consumidor viabilizou que questões de comunicação específicas de algum cliente fiquem isoladas naquele cliente não comprometendo a execução dos demais. Para operacionalizar esta estratégia, lançou-se mão de um atributo na classe do servidor chamado `mascaraQuemSaoOsConsumidores`.

Ao ser incluído um novo consumidor através de uma conexão (tardia ou não) de algum cliente Java, o servidor seleciona um identificador de cliente (a partir da variável `idNovoConsumidor`) e recupera a máscara de *bits* que o identifica realizando uma operação OU com a mascara que identifica os indivíduos atualmente conectados. Quando um cliente vai consumir um recurso do *buffer* compartilhado, ele apresenta a sua máscara de identificação e, se já não consumiu aquele recurso que se encontra no *buffer*, tem permissão de fazê-lo (Figura 5).

Inicialmente o método consumir verifica se existem itens no estoque, ou seja, se o VXt enviou uma informação para o servidor. A função deste método então, é de copiar do buffer compartilhado do servidor para o buffer da *thread* consumidora de cada cliente, que está executando o método consumir.

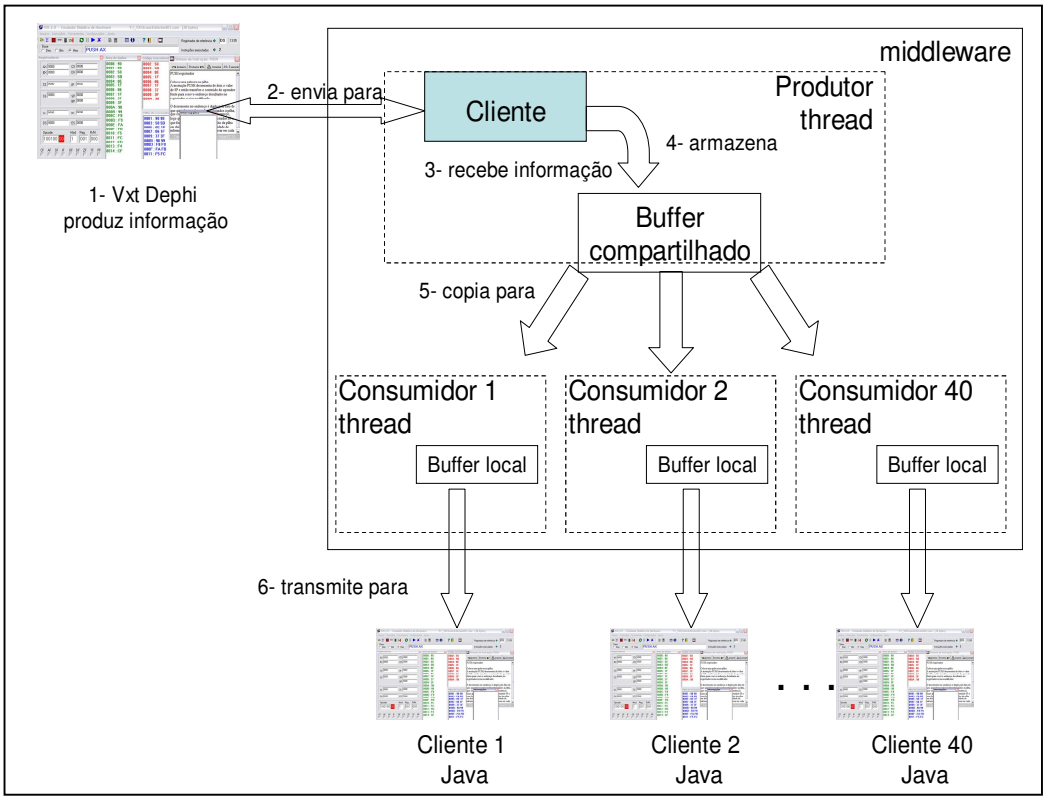

Figura 4 - Arquitetura do middleware

Para isto, executa-se um laço para permitir a este consumidor resgatar todos os recursos produzidos pelo VXt Delphi e copiá-los para o seu *buffer* local. Para isto, varre-se a estrutura de dados do buffer compartilhado, procurando por recursos que possuam na máscara de identificação um bit ligado correspondente ao identificador do consumidor atual. Se o *flag* `temRecursoParaPegar == 0`, significa que todos os consumidores já consumiram o recurso. Então, a função do último consumidor a varrer a lista, é remover aquele recurso do buffer compartilhado.

```
public interface IdentificaClientes {
    public static long codIdConsumidor[] = {
            1,                  //  00000000000000000000000000000001
            2,                  //  00000000000000000000000000000010
            4,                  //  00000000000000000000000000000100
            8,                  //  00000000000000000000000000001000
            16,                 //  00000000000000000000000000010000
            32,                 //  00000000000000000000000000100000
                                . . . . . .
            68719476736l,       //  00010000000000000000000000000000
            1374389534721,      //  00100000000000000000000000000000
            274877906944l,      //  01000000000000000000000000000000
            549755813888l       //  10000000000000000000000000000000
            };
}
```

Quadro1 - Especificação da máscara de bits de identificação das threads consumidores.

Se não houver recurso em estoque, executa uma chamada `estoque.wait()` a qual bloqueia a *thread* até que o servidor informe que há recursos para consumir no buffer compartilhado.

Todo recurso ao ser criado recebe uma máscara de *bits* do servidor

`(mascaraQuemVaiPegarORecurso)` identificando todos os clientes habilitados naquele momento a consumi-lo. Um aspecto a destacar é que, quando as *threads* compartilham acesso a um objeto comum, elas podem entrar em conflito umas com as outras (HORSTMANN, 2004, p. 777). Essa situação é denominada de condição de competição (ou *race condition*) (HORSTMANN, 2004, p. 779).

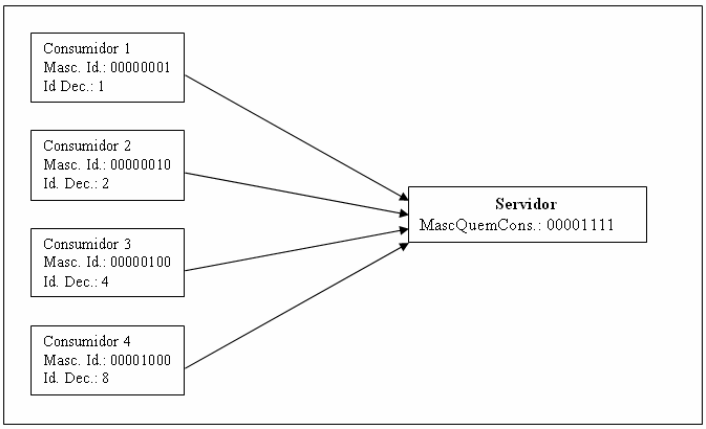

Figura 5 – Organização dos consumidores identificados

Para solucionar o problema, cada *thread* deve ser capaz de bloquear um objeto temporariamente. Enquanto a *thread* bloqueia o objeto, nenhuma outra deve ser capaz de modificar o estado deste objeto. Na linguagem de programação Java utilizam-se métodos sincronizados para bloquear objetos (HORSTMANN, 2004, p. 784). Quando uma *thread* consumiu todos os seus recursos e não tem mais o que fazer, ela chama o método `wait()` liberando o processador para que outras *threads* executem. Quando um novo recurso é adicionado ao *buffer* compartilhado, o método do servidor chama `notifyAll` para desbloquear todas as *threads* consumidoras para que elas resgatem o recurso do buffer global e transmitam para seus respectivos clientes VXt.

3.1 Os meta-forms

Sabendo-se que a aplicação VXt é desenvolvida em Delphi e que as interfaces cliente foram desenvolvidas em Java, havia a necessidade de um protocolo que viabilizasse que alterações nos formulários em Delphi fossem observadas nas interfaces cliente. Assim, desenvolveu-se o conceito de meta-forms - classes que possuem uma cópia dos campos de um formulário e que quando são notificadas da alteração de algum campo, automaticamente enviam para o servidor uma mensagem com a respectiva alteração.

Esta estratégia reduziu significativamente o volume de tráfego na rede produzido entre o VXt em Delphi e os clientes em Java, já que não é necessário que sejam enviadas as informações de todos os campos existentes no sistema a cada ciclo de execução de uma instrução.

3.2 O modelo de contexto

Uma outra característica importante é que a API da biblioteca Softx86 utiliza estruturas de contexto para representar a CPU. Estas estruturas de contexto são declaradas na

aplicação hospedeira (VXt). Com isto é possível emular mais de uma CPU simultaneamente utilizando a mesma biblioteca.

Portanto, com a incorporação da biblioteca Softx86, tornou-se possível implementar um modelo de contexto. Um contexto representa uma instância de uma máquina completa que é executada pelo VXt. Uma instância de contexto compreende:

a) uma instância de *motherboard* a qual, por sua vez, possui uma instância de `TCpu`, `TPic`, `TSO`, `TMemória` e instancias de `TDevice`;
b) `TCpu` é uma classe que viabiliza que o VXt possua uma cópia fiel do conteúdo dos registradores atualizados pela biblioteca Softx86;
c) `TPic` implementa as funcionalidades do *Programmable Interrupt Controller* (PIC);
d) `TSO` implementa um módulo (inicialmente vazio) representando o suporte necessário para a instalação de um sistema operacional na máquina;
e) `TMemória` implementa a memória RAM da máquina;
f) `TDevice` descreve as características principais de um dispositivo de hardware a ser instalado na máquina.

Sendo uma ferramenta didática, é possível que, em determinados momentos, o professor necessite apresentar outros exemplos (além daquele que está sendo utilizado) para caracterizar uma determinada situação. Assim sendo, o VXt permite que várias instâncias de contexto sejam disparadas, com a restrição de que, em determinado momento somente uma estará sendo executada. O modelo de múltiplos contextos possibilita as seguintes variações (MATTOS, et al, 2004):

a) O professor conduz a aula e os alunos acompanham a execução;
b) O professor dispara um trabalho em grupo e pode conectar-se na estação do líder de uma equipe para acompanhar o andamento do trabalho;
c) Cada aluno individualmente pode executar sua própria cópia do servidor e da interface cliente em uma máquina local podendo executar exercícios individuais.

3.3 Execução de uma instrução

A execução de uma instrução envolve uma operação de busca de um *byte* na memória, sua decodificação e efetiva execução. A busca de uma instrução de máquina é apontada pelo par de registradores CS:IP (*code segment*: *instruction point*er). A função **`softx86_decompile_exec_cs_ip(wctx)`** é utilizada para obter uma representação textual da instrução a ser executada para fins de visualização. A função **`softx86_step(wCtx)`** é quem efetivamente executa uma instrução atualizando o contexto de execução da CPU.

Ao ser executada a operação **`softx86_step(wCtx)`**, a biblioteca softx86 lança mão das rotinas de *callback* implementadas em Delphi no VXt para (a) buscar um ou mais bytes na memória do VXt (dependendo do tamanho da instrução e do número de operandos necessários) e (b) para escrever bytes na memória (quando este for o caso). A cada instrução executada com sucesso, uma cópia do conteúdo dos registradores da CPU é obtida pelo VXt de forma a atualizar do formulário específico que apresenta o valor dos registradores ao aluno. Se, após a execução de uma instrução, constatar-se que houve uma interrupção (de software ou de hardware) e, o *flag* de interrupções indica que elas estão habilitadas, então ocorre o *handshake* entre a CPU e o PIC de tal forma a

alterar os valores atuais de CS:IP para o endereço do tratador de interrupções previamente inicializado na tabela de vetores de interrupção (como ocorre no hardware real).

3.4 O acesso à memória do VXt

Conforme citado anteriormente, o acesso à memória RAM do VXt é realizado através de funções *callback*. O quadro 2 apresenta o código do procedimento para acesso de leitura e de escrita à memória do VXt.

Um aspecto que merece destaque no código apresentado no quadro 2 é a necessidade explícita de verificação sobre a tentativa de acesso à memória de vídeo. Caso o acesso à memória seja destinado a faixa de endereços onde reside a memória de vídeo, são chamados os procedimentos **`leCaracterTelaCGA`** e **`escreveCaracterTelaCGA`**, respectivamente, para ler o conteúdo de uma coordenada de tela e escrever o conteúdo em uma coordenada de tela.

```
//Funcao: Callback que retorna um byte de memória ao softx86
//-------------------------------------------------------------------------
procedure pOn_read_memory(_ctx : pointer; address : sx86_udword; var buf : sx86_ubyte; size : Integer); cdecl;
begin
  if address < ctx.GetContextoAtual.ram.InicioMemVideo then
    copymemory(@buf,@ctx.GetContextoAtual.ram._ram[address],size)
  else
    buf := ctx.getContextoAtual.cga.leCaracterTelaCGA(address);
end;

//Funcao: Callback que escreve um byte enviado pelo softx86 na memória do VXt
//-------------------------------------------------------------------------
procedure pOn_Write_Memory(_ctx : pointer; address : sx86_udword; var buf : sx86_ubyte; size : Integer); cdecl;
begin
  if address < ctx.GetContextoAtual.ram.InicioMemVideo then
    copymemory(@ctx.GetContextoAtual.ram._ram[address],@buf,size)
  else
    ctx.getContextoAtual.cga.escreveCaracterTelaCGA(address,buf);
end;
```

Quadro 2 – Funções callback para acesso a memória RAM do VXt.

3.5 O módulo cliente Java

A interface do cliente Java é idêntica a do VXt em Delphi, porém a sua única ação é conectar-se ao servidor. O usuário, em sua interface, somente é capaz de acompanhar a execução do VXt que estará sendo rodado numa outra máquina junto com o servidor.

Toda vez que o VXt aciona a abertura ou fechamento de uma janela, a mesma ação ocorrerá com o cliente, que abrirá ou fechará a janela correspondente em sua interface. As informações que o cliente Java recebe do *middleware*, caso não sejam responsáveis por abrir ou fechar alguma janela, são primeiramente armazenadas em um *ArrayList*, a partir do qual são recuperadas as informações na ordem em que chegaram, atualizando os campos e as janelas da sua interface, sendo posteriormente removida para a utilização da próxima informação que se encontra na fila de dados do *array*. Esse procedimento é necessário para que não se perca nenhuma informação proveniente do *middleware*, numa situação em que o servidor transmita mais rapidamente do que a capacidade do cliente de processar as mensagens recebidas. A conexão entre o cliente Java e o *middleware* acontece através do uso de *sockets*.

3.6 Comunicação via chat

O VXt também possui um recurso de *chat* para viabilizar a comunicação entre os clientes (alunos) e o servidor (professor). Este recurso permite ao servidor enviar mensagens para um cliente em específico, ou para todos os clientes que estejam conectados de uma só vez. O cliente somente pode se comunicar com o servidor.

4 O template para geração de código de sistema operacional no VXt

Até a versão anterior do VXt, a geração de código para testes era realizada através da (a) injeção manual do código executável diretamente na estrutura de dados que representa a memória (na forma de códigos hexadecimais correspondentes) ou (b)através da construção de programas em *assembly*, sua montagem e respectiva geração de código no formato .COM.

Contudo, deve-se destacar que o VXt possui suporte para a carga de programas no formato .EXE. Entretanto, tanto os programas no formato .COM como programas no formato .EXE, em geral, utilizam funções do DOS para a inicialização do ambiente de execução (*run-time*). Este fator sempre limitava a utilização do VXt porque comprometia a execução dos programas exemplo visto que, apesar da biblioteca softx86 implementar o conjunto completo de instruções do processador, a máquina VXt não possuía suporte de sistema operacional – particularmente do BIOS e do DOS. Assim sendo, a partir da primeira chamada de sistema (*system call*) via INT 21H, o programa sendo executado no VXt deixava de executar corretamente.

Tendo em vista superar esta dificuldade, trabalhou-se na concepção de uma solução que permitisse a utilização de um compilador C como ferramenta de desenvolvimento de código para execução pelo VXt e que permitisse o desenvolvimento de aplicações que num ambiente real seriam caracterizadas como embarcado.

Conceitualmente, um sistema embarcado, geralmente, consiste de um microprocessador e um pequeno conjunto de periféricos que executam um programa dedicado. Este programa é armazenado em memória não volátil e geralmente possui limitados recursos de entrada e saída. Como geralmente os sistemas embarcados não possuem um sistema operacional, o programador é responsável por prover todas as funções de entrada e saída de baixo nível. Além disso, diferentemente de programas suportados por um sistema operacional, um sistema embarcado geralmente não é carregado a partir de um disco, mas sim a partir de memórias não voláteis. Conseqüentemente, desenvolver e depurar tais sistemas é uma atividade desafiadora.

Estas características tornam os programas embarcados especialmente interessantes para a aplicação em sala de aula e o VXt é um facilitador neste processo.

Em geral, a utilização de um compilador comercial para o desenvolvimento de aplicações embarcadas requer a alteração nas rotinas de inicialização do ambiente de *run-time* deste compilador de tal forma que todas as chamadas ao BIOS e ao DOS sejam removidas. A solução adotada envolveu a construção de um pequeno núcleo de rotinas, as quais fazem esta função. Deve-se ressaltar que, no caso do VXt não se pretendeu substituir as chamadas ao DOS e ao BIOS tendo em vista que na realidade não existe um DOS nem um BIOS implementados.

O quadro 3 apresenta a parte do *template* de código C utilizado para a geração de código para o VXt.

```
typedef unsigned char far *pointer;
void inicializaTabVetoresInterrupcoesinicializaPonteiros();
void tratadorInterrupcaoSoftware();
void tratadorInterrupcaoSoftware();
void pmonta(pointer *c, unsigned int seg,unsigned int off);
void pdesmonta(pointer c,unsigned int *seg,unsigned int *off);
void main(){
 inicializaTabVetoresInterrupcoes();
 while(1){   asm {nop};   }
}
void  tratadorInterrupcaoSoftware(){
 asm{ //código que salva o contexto em execução
      push  ax;       pushbx;      pushcx;      pushdx;
      push  es;       pushds;      pushsi;      pushdi;
      push  bp
   }
// O código do tratador da interrupção desenvolvido pelo usuário é inserido aqui
 asm{// código que restaura o contexto sendo executado antes do atendimento da interrupção
      pop bp;
      pop di;       popsi;      popds;      popes;
      pop dx;       popcx;      popbx;      popax;
      iret
   }
}
void  tratadorInterrupcaoHardware(){
 asm{
   push   ax;   pushbx;   pushcx;   pushdx;
   push   es;   pushds;   pushsi;   pushdi;
   push   bp
   }
// O código do tratador da interrupção desenvolvido pelo usuário é inserido aqui
 asm{// código que restaura o contexto sendo executado antes do atendimento da interrupção
   pop     bp;
   pop     di;   popsi;   popds;   popes;
   pop     dx;   popcx;   popbx;   popax;
   popf
   iret
   }
}
```

Quadro 3 – Trecho do *template* de código para o VXt.

O quadro 4 apresenta um conjunto de rotinas utilitárias que apresentam o código necessário para a inicialização da tabela de vetores de interrupção apontando para um tratador cujo código é escrito em C.

```
/*----------------------------rotinas utilitárias ----------------------------------------------------
//funcao que monta um pointer a partir de um valor de segmento e offset
void pmonta(pointer *c, unsigned int seg,unsigned int off){
  union{
            pointer addr; unsigned int partes[2];
       }lptr;
  lptr.partes[0] = off;       lptr.partes[1]= seg;      *c =lptr.addr;
}
```

```
//funcao desmonta um pointer em segmento e offset
void pdesmonta(pointer c,unsigned int *seg,unsigned int *off){
union{
          pointer addr;    unsigned int partes[2];
  }lptr;
  lptr.addr = c;              *off = lptr.partes[0];      *seg = lptr.partes[1];
}

/*----------------------------rotinas utilitárias ----------------------------------------------------
//funcao que inicializa os ponteiros da tabela de vetores de interrupção para um tratador exemplo
void inicializaTabVetoresInterrupcoes(){
unsigned int far *ptr;
int i;
unsigned int  seg,off;
  pdesmonta((pointer)&tratadorInterrupcaoHardware,&seg,&off);
  pmonta((pointer)&ptr,0,0x00);
  off += 4; //deslocamento necessário para pular o código gerado pelo turboc
  for (i=0;i<255;i++){
     *ptr=off;
     ptr++;
     *ptr=seg;
     ptr++;
  }
}
```

Quadro 4 – rotinas utilitárias

5 Resultados e discussões

O presente trabalho descreveu o estágio atual do projeto VXt – Virtual XT. O projeto, de caráter didático, consiste na implementação em software de uma máquina baseada no processador 8086. Foram apresentadas as principais características do sistema, sua arquitetura e descrito o *template* para a geração de código de sistema operacional.

As próximas etapas do projeto envolvem o desenvolvimento de dispositivos periféricos e a continuidade no desenvolvimento de código *romable* em linguagem de alto nível. Testes já realizados permitiram o desenvolvimento de programas utilizando o compilador Turbo-C (Borland) e a carga e execução dos mesmos no VXt. Com isto pretende-se viabilizar atividades de desenvolvimento de pequenos sistemas operacionais "embarcados" os quais podem ser analisados em profundidade e de acordo com as características de cada aluno em particular.

Conforme descrito, a propagação da interface para os alunos de uma sala de aula permite ampliar as formas de utilização da ferramenta como recurso de apoio didático. Estão sendo finalizados os preparativos para disponibilizar o software para download no link (www.inf.furb.br /~mattos/vxt). Cabe destacar, também, que este é um projeto de iniciação científica e conta com apoio financeiro do programa PIBIC/FURB.

Referências

CARISSIMI Alexandre S.; OLIVEIRA Rômulo S.; TOSCANI, Simão S. **Sistemas operacionais e programação concorrente**. São Paulo: Sagra Luzzatto, 2003.

CARVALHO, Osvaldo. **GT Middleware.** [S.l.], 2005**.** Disponível em: <http://www.rnp.br/pd/gts2004-2005/middleware.html>. Acesso em: 28 ago. 2006.

CASAS, Luis A. A. **Contribuições para a modelagem de um ambiente inteligente de educação baseado em realidade virtual**. 1999. Tese (Doutorado em Engenharia de Produção) - Departamento de Engenharia de Produção, Universidade Federal de Santa Catarina, Florianópolis. Não paginado. Disponível em: <http://www.eps.ufsc.br/teses99/casas/index.html>. Acesso em: 24 maio 2006.

GOEDERT, Elton. **Propagação da interface do VXt usando o modelo cliente/servidor.** 2006. 60 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

HORSTMANN, Cay. **Big Java**. São Paulo: Bookman, 2004.

LINZMEIER, Marilene. **Tutorial da linguagem Assembly utilizando o VXt**. 1999. 58 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

MATTOS, Mauro M. et al. VXt**:** um ambiente didático para ensino de conceitos básicos de sistemas operacionais e arquitetura de computadores. In: WORKSHOP DE COMPUTAÇÃO DA REGIÃO SUL, 1., 2004, Florianópolis. **Anais…** Florianópolis: Unisul, 2004. Paginação irregular.

SILVEIRA, Janira. **Extensão da ferramenta Delphi2Java-II para suportar componentes de bancos de dados**. 2006. 81 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SOURCEFORGE. **Software x86 CPU emulator library**: help. Version 0.00.0033. [S.l.], 2003. Documento eletrônico disponibilizado com o Softx86. Disponível em: <http://sourceforge.net/search/?type_of_search=soft&words=softx86>. Acesso em: 02 nov. 2006.

VEIGA, Marise S. Computador e educacão? Uma ótima combinação. In: BELLO, José L. P. **Pedagogia em foco**. Petrópolis, 2001. Disponível em: <http://www.pedagogiaemfoco.pro.br/inedu01.htm>. Acesso em: 14 jun. 2006.