# MVX : Uma Ferramenta para Visualização de Programas Multithread

Marnes Augusto Hoff
Mauro Borges da Silva
Celso Maciel da Costa

Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática - Grupo PET Informática
Av. Ipiranga, 6681 – Porto Alegre – RS – Brasil
Tel. +55 51-320 3611 - Fax +55 51 320 2309
e-mail: {marnes, mauro, celso}@andros.pucrs.br

**Resumo:** A sincronização de processos concorrentes é um tópico fundamental no ensino da ciência da computação. Ferramentas gráficas que visualizem os efeitos das operações de sincronização são úteis para se entender seus efeitos e armadilhas. O MVX é uma ferramenta desse tipo. Programas multithreads escritos em C e utilizando a biblioteca Pthreads são instrumentalizados pela ferramenta de Instrumentalização do MVX, que incorpora ao programa funções específicas de monitoração. Quando esses programas são executados, o monitor exibe os efeitos das operações de sincronização e de criação e destruição de threads através de uma animação gráfica.

**Palavras-chave:** Programação concorrente, Programas multithread, Sincronização de processos, Visualização de execução de programas concorrentes, POSIX threads.

## 1 Introdução

A programação multithread é hoje um fator de vital importância dentro da ciência da computação. Através do uso de múltiplas threads de controle para um processo tem-se um ganho de performance, já que há um melhor aproveitamento do processador ocasionado pelo paralelismo das operações.

É importante, contudo, saber que como as threads necessitam se comunicar, o que ocorre através de memória compartilhada, são necessários mecanismos de sincronização para garantir o acesso exclusivo aos dados compartilhados.

Considerando isso, julga-se essencial que os estudantes de computação adquiram não só o conhecimento teórico de como ocorre a execução de programas multithread, mas que também sejam capazes de resolver problemas com o uso desse paradigma de programação. Para tanto, é preciso saber utilizar as funções oferecidas pelo sistema operacional e conhecer o comportamento das mesmas.

Entretanto, mesmo para aqueles aptos a essa tarefa, existe um grande desafio que consiste em testar os programas multithread, pois o indeterminismo de execução torna difícil a depuração destes programas. Esse problema é conseqüência da inadequação da quase totalidade das ferramentas existentes atualmente de tratar com a programação multithread.

Esse trabalho apresenta a ferramenta MVX, que foi desenvolvida no intuito de auxiliar o programador a desenvolver programas multithread. Para isso, o MVX provê a monitoração desses programas através de uma animação gráfica, que permite visualizar a criação e destruição das threads, assim como as operações de sincronização entre elas.

## 2 Programação Multithread

Programação Multithread é uma técnica que permite a um programa executar múltiplas tarefas concorrentemente. O conceito básico de programação multithread já existe há algumas décadas. Contudo, a emergência desse conceito na indústria como um paradigma de programação aceito e padronizado foi um fenômeno dos anos 90[LEW 98].

Essa emergência da programação multithread foi um avanço significativo. Em muitos sistemas operacionais tradicionais, cada processo tem um espaço de endereçamento e uma única thread de controle. Contudo há freqüentes situações nas quais seria interessante ter múltiplas threads de controle dividindo o mesmo espaço de endereçamento e executando praticamente de forma paralela, como se fossem processos diferentes.

Pode-se considerar o exemplo de um servidor de arquivos que fica ocasionalmente bloqueado esperando uma operação de I/O. Caso houvessem múltiplas threads de controle, uma segunda thread poderia ser executada enquanto a primeira aguardasse. Dessa forma tem-se um menor tempo de ociosidade do processador acarretando em uma melhor performance[COU95], [TAN95]. Não é possível, entretanto, obter o mesmo resultado criando dois diferentes processos servidores, pois estes não compartilhariam o mesmo buffer de cache, como ocorre no caso anterior.

Pode-se, ainda, estabelecer outras comparações entre o uso de uma única ou múltiplas threads de controle. Existindo vários processos, cada um tem seu próprio "program counter", sua própria pilha, seu próprio conjunto de registradores e seu próprio espaço de endereçamento. Esses processos, a princípio, não têm nenhuma relação uns com os outros.

No entanto um único processo contendo múltiplas threads apresenta algumas vantagens. Cada thread tem seu próprio "program counter" e pilha; sendo que compartilham a CPU da mesma forma que os processos fazem, ou podem, ainda, ser executadas em paralelo no caso de multiprocessamento. Essas threads podem criar threads filhas e ficar bloqueadas, esperando chamadas de sistema para concluírem, da mesma forma que processos normais. A vantagem é que enquanto uma thread está bloqueada, outra thread, pertencente ao mesmo processo, pode ser executada. Ou seja, da mesma forma que sistemas operacionais multitarefa podem executar mais de um processo concorrentemente, processos podem fazer o mesmo executando mais do que uma thread ao mesmo tempo. Logo, pode-se concluir que uma thread representa para um processo o mesmo que este representa para uma máquina[LEW98], [TAN95].

Cada thread consiste em um fluxo de controle diferente, que pode executar suas instruções de forma independente, permitindo a um processo multithread executar várias tarefas de forma concorrente.

Certamente com a programação multithread tem-se um grau maior de complexidade. Diferentes threads em um processo não são tão independentes quanto processos diferentes, pois compartilham o mesmo espaço de endereçamento, as mesmas variáveis globais, o conjunto de arquivos abertos, timers, etc. Como cada thread pode acessar qualquer endereço virtual, uma thread poderia, até mesmo, ler e escrever na pilha de outra thread[TAN95]. Ou seja, não existe uma proteção entre threads, até porque elas devem cooperar entre si e não se confrontar. De qualquer forma torna-se uma questão mais complexa de ser controlada e visualizada pelo programador, pois não há uma ferramenta eficaz de depuração na qual se basear.

Todos os sistemas operacionais modernos (Windows 95, Windows NT, OS/2, Solaris, etc.) suportam programação multithread.

## 2.1 Por que utilizar threads?

Há, na verdade, apenas uma razão para se utilizar threads: obter programas melhores e mais rápidos[LEW98]. No entanto pode-se listar outros benefícios propiciados pelos programas multithread:

- Paralelismo: computadores com mais de um processador oferecem um grande potencial para o desenvolvimento de aplicações mais rápidas. Programação Multithread é uma forma eficiente para que os desenvolvedores de aplicações explorem o paralelismo do hardware. Diferentes threads podem ser executadas em diferentes processadores simultaneamente.

- Melhor aproveitamento do processador: mesmo em máquinas com um único processador o uso de threads traz vantagens. Quando um programa tradicional requisita um serviço ao sistema operacional, ele precisa esperar até que o serviço tenha sido completado para continuar sua execução. Com o uso de threads, enquanto uma thread do programa espera por uma operação de entrada e saída, por exemplo, outra thread do mesmo programa pode ocupar o processador.

- Recursos do sistema: programas que usam dois ou mais processos para acessar dados comuns em uma memória compartilhada estão efetivamente utilizando mais de uma thread de controle. Contudo, cada um dos processos precisa manter uma estrutura completa, incluindo um espaço de memória virtual e estado do kernel. Isso tudo faz com que cada processo tenha um custo muito grande em termos de tempo e espaço, em relação ao uso de threads. Threads utilizam apenas uma fração dos recursos de sistema necessessarios a um processo.

- Código fonte portável: muitas vezes um programa necessita poder ser executado em diferentes plataformas. Como uso de POSIX Threads (padrão), é possível escrever um único código fonte e apenas recompilá-lo para diferentes plataformas.

## 2.2 Sincronização entre Processos

A sincronização garante que múltiplas threads de controle coordenem suas atividades de forma que uma thread não modifique, acidentalmente, os dados nos quais outra thread esteja trabalhando. Isso é proporcionado através do uso de funções responsáveis por limitar o número de threads que podem acessar determinados dados concorrentemente.

No caso mais simples somente uma thread pode executar um dado trecho de código de cada vez. Esse código, possivelmente, altera dados globais ou faz leitura e escrita em um dispositivo. Por exemplo, se uma thread T1 obtém um lock e começa a trabalhar com dados globais, uma thread T2 deverá aguardar até que a primeira termine para que possa executar o mesmo código. Assim, pode-se garantir que os dados continuem consistentes[LEW 96].

### 2.3 POSIX Threads

POSIX Threads, ou Pthreads, é um novo padrão POSIX API que provê uma biblioteca de rotinas utilizadas para o desenvolvimento de programas multithread. A ferramenta, descrita neste artigo, tem a tarefa de monitorar a execução de funções dessa biblioteca, relativas à sincronização entre threads[LEW96].

## 3 Monitoração de Programas Multithread

Programas escritos em UNIX C, utilizando programação multithread, podem ser automaticamente animados, no que se refere às suas operações de criação e sincronização de threads, pela ferramenta descrita a seguir. O funcionamento da ferramenta de visualização é baseado em dois módulos. A figura 1, a seguir, ilustra o pré-processamento e posterior execução de um programa multithread sendo monitorado com o uso da ferramenta implementada.

O primeiro módulo é responsável pela instrumentalização, isto é, ele insere chamadas de funções específicas dentro do código fonte de um programa multithread escrito em UNIX C. O código fonte resultante dessa instrumentalização será compilado por um compilador UNIX C normalmente. Quando o programa for executado, as funções inseridas enviarão as informações relativas à sincronização das threads, necessárias à animação, para o segundo módulo.

Ele gera, também, um arquivo HTML onde estará incluída uma chamada para o segundo módulo, que é um applet JAVA, com os parâmetros necessários para sua posterior execução .

O segundo módulo é um processo daemon que aceita mensagens de um processo instrumentalizado, através de um canal de comunicação (socket), interpretando-as de forma a gerar uma saída gráfica em JAVA, que é exibida por um Appletviewer.

Para auxiliar no processo de depuração, pode-se visualizar a animação passo-a-passo.

## 4 Módulo de Instrumentalização

Esse módulo tem como função, inicialmente, analisar um código fonte original qualquer, em UNIX C, reconhecendo as chamadas de sistema relevantes à sincronização de threads. Além disso ele ainda gera uma página HTML com a inclusão do monitor de execução (applet JAVA), com os parâmetros necessários à sua execução. Inicialmente, ao executar o instrumentalizador no arquivo

desejado, deve-se escolher o nome do host e número da porta onde será executado o monitor de execução (sendo o número da porta incluído como parâmetro na página HTML).

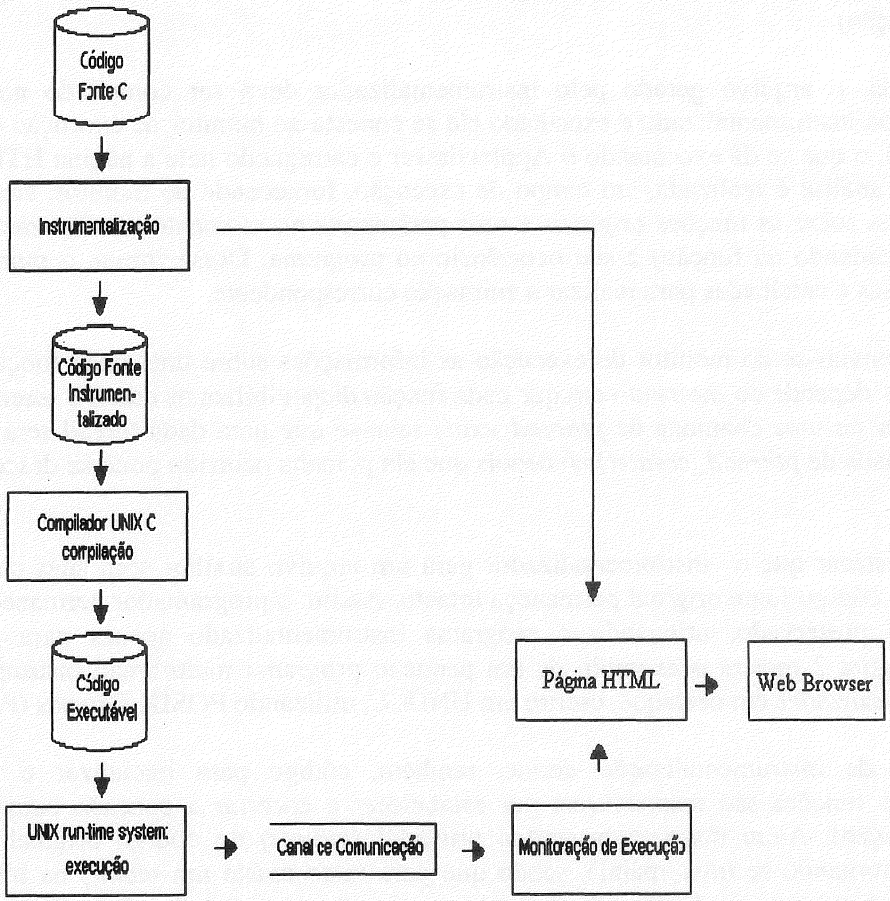



**Figura 1** – Fases da animação de um programa multithread

Quando executado, o instrumentalizador começa o processo de análise do código fonte. Conforme vai analisando o código, ele vai copiando-o para um arquivo auxiliar que apresenta o mesmo nome do arquivo original precedido pelo prefixo "mvx_". Durante esse processo, toda vez que for encontrada um função considerada relevante, esta será copiada para o arquivo auxiliar, sendo também precedida pelo prefixo "mvx_".

Essas funções precedidas constam de uma biblioteca de funções que o instrumentalizador insere como *#include* no arquivo auxiliar, sendo elas as responsáveis pela análise das funções originais e envio de informações relativas às mesmas ao monitor de execução, que é o segundo módulo. O instrumentalizador cria ainda um outro arquivo que é a página HTML com a chamada do applet JAVA (monitor de execução).

Em seguida, o arquivo gerado pelo instrumentalizador deve ser compilado normalmente. Quando o programa instrumentalizado é executado ele se conecta ao monitor de execução que já deve estar em execução, o que se dá executando o Appletviewer e carregando nele a página HTML gerada. Em seguida, uma análise é realizada, em tempo de execução, fornecendo ao monitor, através de um socket, informações sobre as funções originais e seus parâmetros no momento imediatamente anterior ou posterior (dependendo da função) à sua ocorrência no programa. Dessa forma, o monitor recebe informações precisas e detalhadas para realizar a animação correspondente.

O fato de enviar-se ao monitor de execução as informações sobre uma dada função antes ou depois dela ocorrer depende do momento em que cada função disponibiliza os dados a serem enviados. Por exemplo, antes de uma chamada de *pthread_exit()* sabe-se que uma dada thread será terminada, mas em uma chamada de *pthread_create()* só depois que ela já tenha ocorrido pode-se descobrir qual é o seu PID.

Deve-se destacar que o instrumentalizador gera um arquivo auxiliar com suas chamadas de sistema para que o código fonte original permaneça intacto. Assim, o programador permanece com seu programa original preservado, utilizando o programa instrumentalizado apenas para executar a monitoração. A figura 2 mostra o exemplo de um pequeno programa multithread instrumentalizado com as alterações realizadas em destaque, escrito em UNIX C, utilizando POSIX Threads (Pthreads).

O módulo de instrumentalização define, também, código para inicializar e finalizar a monitoração. Essas funções são responsáveis por estabelecer e encerrar a conexão com o servidor (monitor de execução). Além disso, o programa principal (*main()*) do código original também é instrumentalizado tornando-se mvx_main(), sendo que para tanto existe um *main()* na biblioteca de funções da ferramenta (onde também estão incluídas as funções de inicialização e finalização) que tomará a posição de programa principal. O novo *main()* tem como tarefas estabelecer a conexão com o monitor de execução, executar o *mvx_main* e, por último, encerrar a conexão.

Entretanto, para que após a instrumentalização o programa não perca a sua precisão, a chamada de sistema original e seu respectivo código de análise devem ser executados de forma atômica e não intercalada com a execução de outras chamadas de sistema instrumentalizadas. Para tanto, a instrumentalização garante a exclusão mútua dessas execuções através do uso de semáforos, tornando-as atômicas.

```
#include <pthread.h>
#include <stdio.h>
#include "mvx.h" // Biblioteca de funções do MVX

mvx_pthread_t tid1,tid2;

void * hello(){

puts("Hello World");

}

mvx_main()
{
  int result, result2;

  mvx_pthread_create(&tid1, NULL, hello, NULL);
  mvx_pthread_create(&tid2, NULL, hello, NULL);
  mvx_pthread_join(tid1, NULL);
  mvx_pthread_join(tid2, NULL);

}
```

**Figura 2 –** Exemplo de um programa instrumentalizado

## 5 Monitor de Execução

Esse módulo é um processo daemon que gera uma animação gráfica a partir das informações recebidas do primeiro módulo. Essa animação é criada através do uso de funções JAVA, sendo possível fazer sua visualização em um Appletviewer local ou remoto executado em uma máquina UNIX.

O monitor de execução deve ser executado em uma determinada máquina (que será a máquina que executa o Appletviewer) e porta, podendo o programa instrumentalizado se conectar a ele dependendo apenas das opções feitas no momento da instrumentalização do programa fonte original.

O monitor de execução possui uma interface usada para a comunicação com o programa instrumentalizado, recebendo dados detalhados gerados pelas funções de análise do instrumentalizador. Essa comunicação, realizada através de sockets, informa ao monitor sobre todos os eventos relevantes que vão ocorrendo durante a execução. Ele também é capaz de realizar a execução de um programa passo-a-passo, possibilitando um melhor entendimento do seu comportamento. Essa interface foi desenvolvida em C.

Ao receber esses dados, um outro módulo, também desenvolvido em C, faz o tratamento dos mesmos. Esse tratamento envolve a transformação dos dados recebidos em primitivas básicas para a animação, realizada pelo terceiro módulo.

O terceiro módulo liga essas informações recebidas e tratadas à geração do gráfico, realizada em JAVA. Ou seja, as informações recebidas pelo módulo de tratamento dos dados, desenvolvido em C, são passadas para o módulo desenvolvido em JAVA que executa a função gráfica correspondente. Para possibilitar essa interação, o monitor de execução é, na verdade, um applet JAVA que utiliza métodos nativos implementados em C. A própria linguagem JAVA possui mecanismos para permitir essa interação com a linguagem C. Os casos que levam a se fazer essa interação são, basicamente: necessidade de melhor desempenho já que JAVA não é suficientemente rápida para aplicações de tempo crítico e aplicativos que precisam utilizar características do sistema que não são supridas pelas classes JAVA[ARN 97].

Pode-se dividir o monitor de execução em três módulos, como representado graficamente na figura 3:

- Módulo 1 (Linguagem C) – controla a comunicação pelo socket. Encaminha as informações recebidas para o módulo 2.
- Módulo 2 (Linguagem C) – recebe informações do módulo C, tratando-as e transformando-as em primitivas para a animação, realizada pelo módulo 3 (JAVA). As primitivas geradas são armazenadas em um buffer.
- Módulo 3 (Linguagem JAVA) – módulo que busca as primitivas geradas pelo módulo 2, retirando do buffer as primitivas geradas pelo módulo 2, fazendo uso de funções gráficas para gerar componentes básicos para a animação disparada no Appletviewer, representando as operações sobre threads.

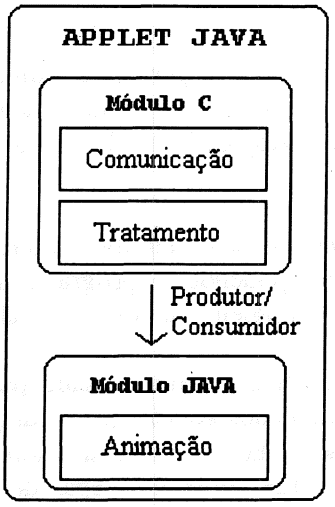



**Figura 3** – Estrutura do Monitor de Execução

## 6 Componentes Gráficos e Animação

A animação gráfica gerada pelo Monitor de Execução contém os componentes apresentados na figura 4, a seguir:

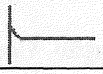
| | |
|---|---|
| Ícone 1  | A criação de uma thread é representada pelo ícone 1, sendo mostrado ao seu lado o identificador da thread. * |
| Ícone 2  | Uma thread em execução é representada pelo ícone 2. * |
| Ícone 3  | Uma thread bloqueada, tanto no caso de um **join** como no caso de um **mutex**, é representado pelo ícone 3. * |
| Ícone 4  | O término, implícito ou explícito, de uma thread é representado pelo ícone 4. * |
| Ícone 5  | Quando uma thread efetua um **join**, ela fica bloqueada (ícone 3) esperando o término da thread alvo e surge ao seu lado o ícone 5. Quando a thread alvo termina a thread que efetuou o join é desbloqueada (ícone 2). ** |
| Ícone 6  | O movimento de uma thread é representado pelo ícone 6. |
| Ícone 7  | O ícone 7 representa um **mutex** bloqueado. *** |
| Ícone 8  | O ícone 8 representa um **mutex** não bloqueado. *** |
| Ícone 9  | O ícone 10 representa um **mutex** unlock. *** |
| Ícone 10  | Ao criar uma thread filha, surge uma bifurcação representada pelo ícone 11. |

**Figura 4** – Componentes da animação gráfica

* Os ícones 1, 2, 3 e 4 possuem , ao seu lado, o identificador da thread.
** O ícone 5 possui, ao seu lado, o identificador da thread alvo.
*** Os ícones 7, 8 e 9 possuem, ao seu lado, o identificador do mutex.

Sempre que um mutex é criado, este é mostrado ao lado da animação da execução, sendo possível sempre visualizar todos os mutex ativos e seu respectivo estado (bloqueado ou desbloqueado).

Para se exemplificar a saída gráfica gerada pelo MVX, considere-se o exemplo de código abaixo:

```
void * x () // Thread 4
{
      ...
      pthread_mutex_lock(...); // Faz lock no mutex 44
      ...
      pthread_mutex_unlock(...); // Faz unlock no mutex 44
      ...
}
main() // Thread 1
{
      ...
      pthread_mutex_init(...); // Cria mutex 44
      ...
      pthread_create(...); // Cria thread 4
      ...
      pthread_mutex_lock(...); // Faz lock no mutex 44
      ...
      pthread_join(...); // Faz join na thread 4
      ...
      pthread_mutex_destroy(...); // Destrói mutex 44
      ...
}
```

Durante a execução desse código o MVX irá gerar uma animação gráfica, representada pela figura 5. Cada linha da animação gerada corresponde a um dos passos que são apresentados abaixo:

1. Criação da thread 1.
2. Thread 1 em execução.
3. Thread 1 em execução enquanto ocorre a criação do mutex 44.
4. Thread 4 é criada pela thread 1.
5. Threads 1 e 4 em execução e mutex 44 ativo e desbloqueado.
6. Thread 1 faz lock no mutex 44 e consegue, enquanto a thread 4 continua em execução. Com isso a representação dos mutex ativos mostra que agora o estado do mutex 44 é bloqueado.
7. Threads 1 e 4 em execução e mutex 44 ativo e bloqueado.
8. Thread 1 faz join na thread 4 e fica bloqueada, enquanto a thread 4 continua em execução e o mutex 44 ativo e bloqueado.

9. Thread 1 continua bloqueada no join esperando o término da thread 4. Thread 4 faz lock no mutex 44 e não consegue, pois a thread 1 está com o lock desse mutex no momento, sendo assim a thread 4 fica bloqueada. Como a thread 1 continuará bloqueada até que a thread 4 termine e a thread 4 continuará bloqueada até que a thread 1 continue sua execução e libere o lock no mutex 44, temos um Deadlock.

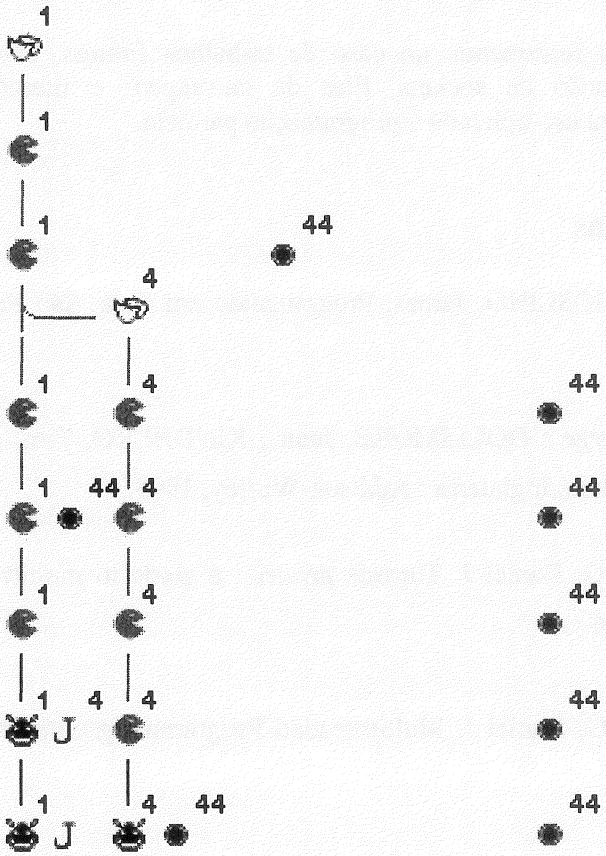



**Figura 5** – Animação de um exemplo de Deadlock

## 7 Conclusão

A ferramenta MVX, descrita nesse artigo, tem como função animar automaticamente programas multithread UNIX C (que utilizam POSIX Threads), quanto às suas funções de sincronização.

Com isso, objetiva-se seu uso no ensino de ciências da computação, principalmente em aulas prática de sistemas operacionais, já que há uma considerável demanda por novos métodos de ensino interativos nessa área.

Outro objetivo pretendido é o auxílio no desenvolvimento de programas multithread, facilitando a depuração através da visualização de uma animação gráfica representando problemas de sincronização.

Um ponto importante a considerar é a escolha da biblioteca Pthreads para a monitoração, que foi feita por ser esta definida como padrão. A ferramenta implementada apresenta, ainda, uma interface amigável, incorporando um visualizador que permite monitoração remota do programa.

Como extensão para a ferramenta, no caso de trabalhos futuros, existem algumas opções interessantes, como: monitoração de sockets, filas de mensagens e memória compartilhada e, naturalmente, sua extensão para uso aplicado à programação paralela.

## 8 Referência Bibliográficas

[ARN 97] ARNOLD, Ken ; GOSLING, James. Programando em Java. São Paulo : Makron Books, 1997.

[COU 95] COULOURIS, George ; DOLLIMORE, Jean ; KINDBERG, Tim Distributed systems : concepts and design. Workingham, Inglaterra : Addison-Wesley, 1995.

[LEW 96] LEWIS, Bil ; BERG, Daniel J. Threads primer : a guide to multithreaded programming. New Jersey : Prentice-Hall,1996.

[LEW 98] LEWIS, Bil ; BERG, Daniel J. Multithreaded Programming with Pthreads. New Jersey : Prentice-Hall, 1998.

[TAN 95] TANENBAUM, Andrew S. Distributed operating systems. Prentice-Hall, Englewood Cliffs, 1995.

[VOG 97] VOGT, Carsten. Visualizing UNIX Synchronization Operations. Cologne Polytechnic. Cologne, Germany.