

Máquina Virtual (MV)

Cleyson Oliveira *, Gustavo Duarte †, Ismael Vargas ‡, Rafael Cardoso §

Pontifícia Universidade Católica do Rio Grande do Sul

PUCRS - Escola Politécnica - Engenharia de Software

1 de dezembro de 2020

Resumo

Este artigo abordará a resolução do primeiro problema da disciplina de Sistemas Operacionais - SisOp - ministrado pelo professor Fernando Luis Dotti em 2020/2 para a turma 127. Descrevendo as tecnologias e ferramentas utilizadas para o desenvolvimento do projeto da "Máquina virtual"(MV) com suas soluções e manuais de instrução.

Este desenvolvimento terá como objetivo a compreensão do funcionamento interno da CPU (Central Process Unit), seus registradores e memória, de uma maneira segura e simples, fazendo a sua execução e visualização protegidas pelas ferramentas de segurança já existentes nos sistemas e linguagens de alto nível atuais.

Introdução

A proposta é apresentada como participação da nota de avaliação na cadeira 4647D-04, onde os alunos, em grupos de até quatro participantes, serão responsáveis por desenvolver, em linguagem de alto nível, um pequeno programa capaz de simular o funcionamento de um processador, mostrando a interação com seus registradores e memória.

O processador em questão terá à disposição um registrador para instruções (IR) e outros oito registradores (R0 a R7), para a execução dos programas em *Assembly*. O mesmo também possuirá o *Program Counter* (PC), este que é responsável por controlar a posição da memória em que a máquina virtual estará interagindo.

A memória será representada através de um *array* contíguo de n posições definida nas propriedades da máquina virtual. Cada posição da memória terá capacidade para comportar 4 bytes, codificados em [OPCODE, 1 REG 0..7, 1 REG 0..7, Parâmetro K ou A conforme operação].

*Cleyson.Oliveira@edu.pucrs.br

†gustavo.hernandez@edu.pucrs.br

‡ismael.vargas@edu.pucrs.br

§rafael.santos00@edu.pucrs.br

Tecnologias

A seleção de tecnologias foi realizada levando como critério principal a legibilidade e facilidade de execução do código, desconsiderando conceitos como o desempenho e a velocidade de execução ou eficiência em uso de memória. Sendo o principal objetivo a acessibilidade na execução de códigos *Assembly* na máquina virtual, sem que os parâmetros emulados de baixo nível precisem ser levados em consideração.

0.1 Linguagem

O projeto será feito com base na linguagem script Groovy[2], a qual é desenvolvida para plataformas *JAVA (JVM)*, sendo fortemente utilizada no framework Grails, este que utiliza do framework *Java Spring Boot*. Sua escolha deu-se devido à facilidade de organização de suas funções, onde é permitido chamar os métodos da classe como uma variável *String*, assim permitindo a melhor organização dentro dos pacotes de desenvolvimento.

0.2 Gerenciador de pacotes

Os pacotes serão gerenciados pelo *Gradle*, o qual é um sistema de automação de compilação *open source* que se baseia nos conceitos de *Apache Ant* e *Apache Maven*, porém, utiliza tecnologias mais recentes, como o Groovy, para substituir a declaração de configuração *XML*, conforme utilizado no *Maven*.

Esta ferramenta foi escolhida como gerenciador pois há um maior suporte para a linguagem escolhida; outro ponto é que a sua declaração de configuração é muito mais limpa e legível comparado com o *Maven*, possibilitando, desta forma, a fácil criação de *tasks* para a automação de outras partes do processo, como por exemplo, a criação do artefato de entrega.

0.3 Dependências

1. Codehaus Groovy (v3.0.5) [JVM Languages]: Referência ao compilador do projeto;
2. Junit jupiter api (v5.6.2) [Testing Frameworks]: Usado nas escritas dos códigos e extensões;
3. Junit jupiter engine (v5.6.2) [Testing Frameworks]: [*Runtime*] implementação da *engine*.

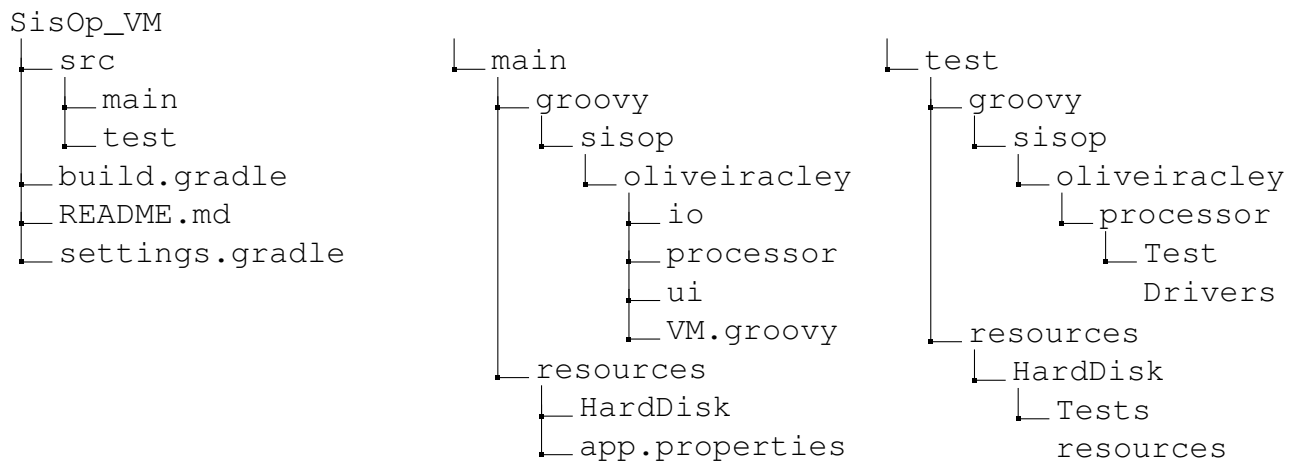
0.4 Manual de utilização

Para executar qualquer comando no projeto é necessário abrir um terminal (*shell*) dentro da pasta raiz. Os comandos a seguir são todos os necessários para utilizar o programa.

1. *gradle run* - Executa o projeto
2. *gradle test* - Executa os testes no projeto
3. *gradle clean* - Exclui os arquivos de compilação
4. *gradle artifact* - Gera o artefato de entrega
5. *gradle run - -args="FileName"* - Executa o projeto passando o arquivo por parâmetro

0.5 Estruturação do projeto

O *Gradle* se responsabilizará por toda a estruturação, compilação e organização dentro da pasta raiz do projeto (*SisOp_VM*). Conforme a seguinte árvore:

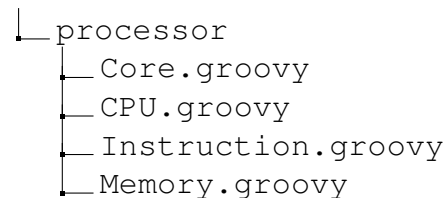


1 Desenvolvimento

No pacote raiz do desenvolvimento (*sisop.oliveiracley*), temos apenas a classe *VM.groovy*, qual contem o método *main*, nos permitindo indicar o início do programa. Assim como um projeto em *gradle*, sendo ele *Spring Boot* ou *Grails*, a classe *main* é exclusiva e usada unicamente para chamar a classe que inicializará todo o programa.

Diferente dos projetos padrões, esta classe conta com uma variável *String* pública e final, qual indica o caminho para o *application.properties*, que é o arquivo de configuração da máquina virtual.

Dentro do pacote *sisop.oliveiracley.processor*, é onde toda a parte importante do emulador acontece. A proposta foi realizar um desenvolvimento que demonstrasse separadamente cada função da máquina virtual, diferenciando então o *OpCode* da memória e da *CPU*.



Core, representa a descrição e execução de todos os *OpCodes*. Esta classe é como uma extensão da *CPU*, seu intuito é tornar os *OpCodes* legíveis e organizados.

O *CPU* é responsável pelo *FETCH* com a instrução da memória e também a parte da repetição até o final do programa *Assembly* que estiver em execução.

A classe *Instruction* reserva-se à estrutura de como cada posição da memória será organizada e alocada, definindo assim o tamanho e informações que trafegam pelo processador.

Por último, a classe *Memory*, é responsável por armazenar todos os códigos *Assembly* a serem executados e também os dados gerados pelos programas. A mesma também tem a função de gerenciar a alocação dos programas em todas as suas posições.

1.1 Gerenciamento de memória

O sistema de gerenciamento de memória é feito através de páginas e frames. Durante o momento de alocação da memória, a máquina virtual passa em todas as páginas até que o espaço total do programa a ser carregado caiba no espaço que está sendo alocado. As posições de memória funcionam de maneira 'virtual', onde, para a *CPU*, a memória sempre será de 0 até o tamanho solicitado para o programa durante a alocação, para o seu acesso, a memória fará o *bind* pelo nome do programa (*id*) que será armazenado em uma estrutura *map* que indica em quais páginas o mesmo se apresenta, consequentemente, à quais *frames* da memória que o programa pode acessar.

Caso não seja possível esse acesso, dado a presença de outro programa, a interrupção *Invalid Address* é lançada à *CPU*. Também outras validações como espaço disponível na memória e programas duplamente carregados são previstas pelo gerente de memória.

1.2 Gerenciador de processos

O gerente de processos emprega o escalonador *Round-Robin* de *quantum* '5', sendo, nesse caso, o quantum a representação de cinco *words*, palavras, interpretadas pelo *CPU*. O escalonador é formado por três estruturas. Uma delas sendo uma fila *FIFO*, de processos prontos para rodar, e as outras duas são listas encadeadas (*LinkedList*) de processos bloqueados e processos terminados.

A fila de processos prontos para execução é ordenada de acordo com a prioridade do processo. Todos os processos criados começam com prioridade alta e a cada execução os mesmos tem a sua prioridade reduzida. Sendo assim, ao começar as execuções em paralelo, o gerenciador de processos dará preferência (em ordem de inserção), mas também respeitando a prioridade dos processos, isso é, primeiro são executados todos os processos de prioridade alta, para então executar os de prioridade inferior e assim por diante.

1.2.1 Processos bloqueados em IO

Como todos os processos rodam em paralelo, ao encontrar a instrução *TRAP*, qual é responsável por fazer a solicitação de *IO*, o processo ficará preso na lista de bloqueado do gerente de processos até que o *console* seja respondido e o processo liberado de volta para a lista de processos rodando.

2 Manual da Máquina Virtual

2.1 Programas assembly

Os arquivos devem seguir algumas regras básicas para a sua leitura correta. os programas devem ser salvos em arquivos sem extensão, por exemplo "Assembly_Fibonacci". Para solicitar alocação de memória para o gerente de memória, basta escrever "___" no código que será considerado como espaço de memória em branco de domínio do programa. Atente que no final do terceiro *underline* sucede um espaço em branco, assim como a instrução *STOP*.

O arquivo deve terminar com uma linha vazia, veja como exemplo o subtópico '2.1.2 Exemplo de programa', linha 6.

2.1.1 Instruções aceitas pela MV

OPCODE	Syntax	Micro-operation	R1	R2	P	Description
JMP	JMP k	$PC \leftarrow k$			K	Direct jump
JMPI	JMPI Rs	$PC \leftarrow Rs$	Rs			Register jump
JMPIG	JMPIG Rs, Rc	$Rc > 0 ? PC \leftarrow Rs : PC++$	Rs	Rc		Jump if greater
JMPIL	JMPIL Rs, Rc	$Rc < 0 ? PC \leftarrow Rs : PC++$	Rs	Rc		Jump if less
JMPIE	JMPIE Rs, Rc	$Rc = 0 ? PC \leftarrow Rs : PC++$	Rs	Rc		Jump if equal
JMPIM	JMPIM [A]	$PC \leftarrow [A]$			A	Memory jump
JMPIGM	JMPIGM [A], Rc	$Rc > 0 ? PC \leftarrow [A] : PC++$		Rc	A	Memory jump if greater
JMPILM	JMPILM [A], Rc	$Rc < 0 ? PC \leftarrow [A] : PC++$		Rc	A	Memory jump if less
JMPIEM	JMPIEM [A], Rc	$Rc = 0 ? PC \leftarrow [A] : PC++$		Rc	A	Memory jump if equal
ADDI	ADDI Rd, k	$Rd \leftarrow Rd + k$	Rd		K	Immediate addition
SUBI	SUBI Rd, k	$Rd \leftarrow Rd - k$	Rd		K	Immediate subtraction
LDI	LDI Rd, k	$Rd \leftarrow k$	Rd		K	Load immediate
LDD	LDD Rd,[A]	$Rd \leftarrow [A]$	Rd		A	Load direct from data memory
STD	STD [A],Rs	$[A] \leftarrow Rs$	Rs		A	Store direct to data memory
ADD	ADD Rd, Rs	$Rd \leftarrow Rd + Rs$	Rd	Rs		Addition
SUB	SUB Rd, Rs	$Rd \leftarrow Rd - Rs$	Rd	Rs		Subtraction
MULT	MULT Rd,[Rs]	$Rd \leftarrow Rd * Rs$	Rd	Rs		Multiplication
LDX	LDX Rd,[Rs]	$Rd \leftarrow [Rs]$	Rd	Rs		Indirect load from memory
STX	STX [Rd],Rs	$[Rd] \leftarrow Rs$	Rd	Rs		Indirect storage to memory
SWAP	SWAP Ra, Rb	$T \leftarrow Ra; Ra \leftarrow Rb; Rb \leftarrow T$	Ra	Rb		SWAP registers
STOP						STOP (HALT)
CONF						Configure CPU output
DATA					D	Memory address has data
___						Memory address is empty
TRAP	TRAP k, [Rs]	$k=1 ? [Rs] \leftarrow [IO] : IO \leftarrow [Rs]$	Rs		K	Call IO request

2.1.2 Exemplo de programa

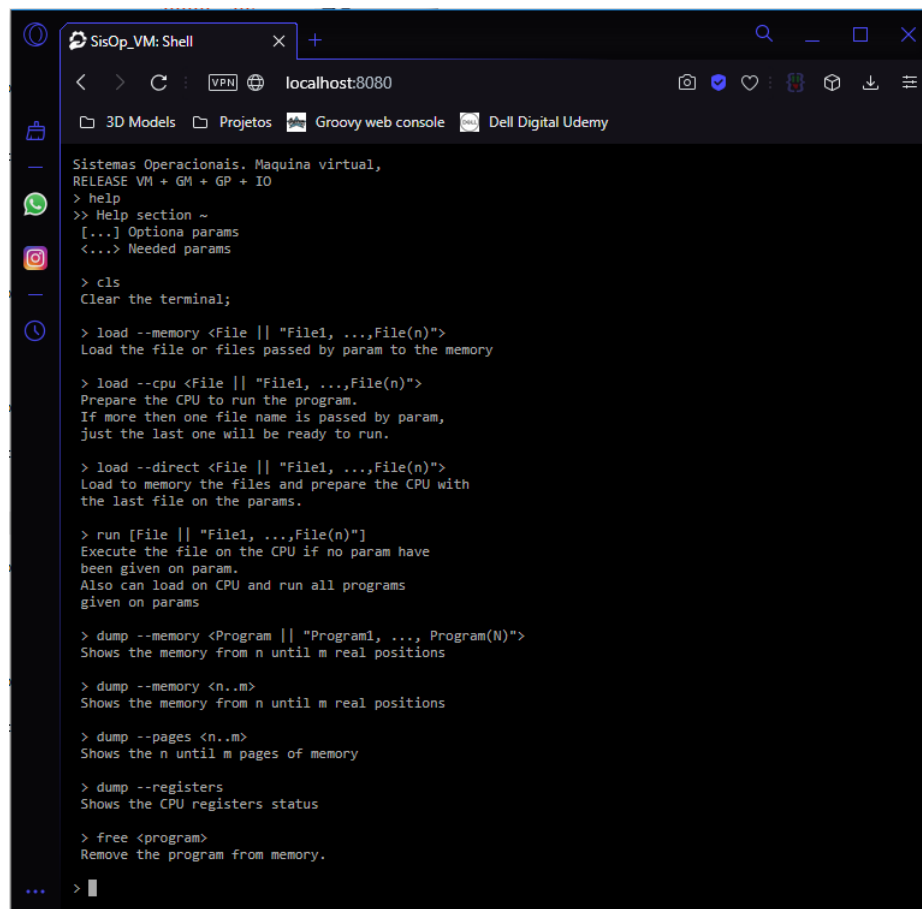
```

1  LDI R1, 171
2  STD [4], R1
3  TRAP 2, 3
4  STOP
5  ___
6

```


2.2.3 Interface Shell

Para utilizar o *Shell* da Máquina Virtual, é necessário abrir o navegador na url *localhost:8080/shell* ou simplesmente *localhost:8080*. A mesma funciona como um terminal propriamente dito, todas as suas funcionalidades são descritas ao executar o comando *help*.



```
Sistemas Operacionais. Máquina virtual,  
RELEASE VM + GM + GP + IO  
> help  
>> Help section ~  
[...] Optiona params  
<...> Needed params  
  
> cls  
Clear the terminal;  
  
> load --memory <File || "File1, ...,File(n)">  
Load the file or files passed by param to the memory  
  
> load --cpu <File || "File1, ...,File(n)">  
Prepare the CPU to run the program.  
If more then one file name is passed by param,  
just the last one will be ready to run.  
  
> load --direct <File || "File1, ...,File(n)">  
Load to memory the files and prepare the CPU with  
the last file on the params.  
  
> run [File || "File1, ...,File(n)"]  
Execute the file on the CPU if no param have  
been given on param.  
Also can load on CPU and run all programs  
given on params  
  
> dump --memory <Program || "Program1, ..., Program(N)">  
Shows the memory from n until m real positions  
  
> dump --memory <n..m>  
Shows the memory from n until m real positions  
  
> dump --pages <n..m>  
Shows the n until m pages of memory  
  
> dump --registers  
Shows the CPU registers status  
  
> free <program>  
Remove the program from memory.  
... > |
```

Figura 3: Terminal shell, utilizando o programa

2.2.4 Interface Console

O terminal *console* que mostra os processos bloqueados está presente na url *localhost:8080/console*. O mesmo tem sua funcionalidades simples, onde o botão *free* libera o processo de escrita e ao digitar o valor na caixa do processo de *read* e pressionar a tecla *enter*, o mesmo é executando.

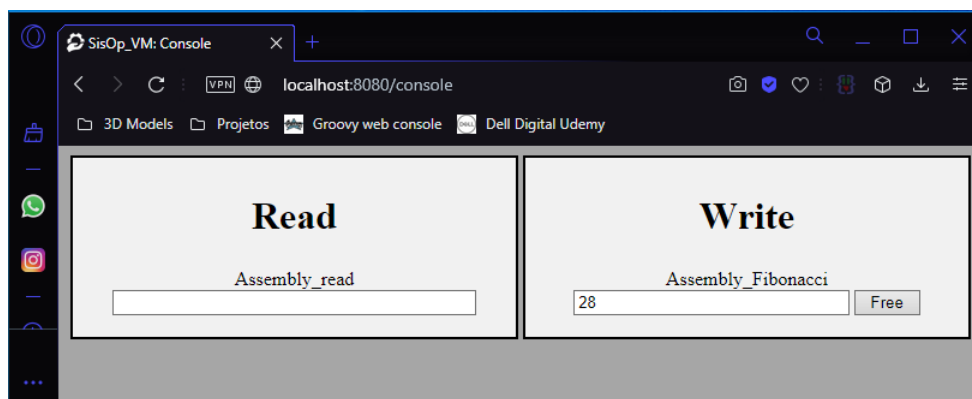


Figura 4: Terminal Console, lista de processos bloqueados

Conclusão

Este trabalho não teve um peso avaliativo tão grande quanto o valor educacional. Durante seu desenvolvimento, percebeu-se vários gaps de informação que acreditava-se ter sido compreendido, porém, erroneamente, com a utilização prática das técnicas apresentadas em aula, a Máquina Virtual se mostrou um forte aliado e um belo parque de diversões para consolidar todo esse conhecimento.

Um único ponto que gostaríamos de ressaltar como um adendo à esta metodologia é a posição do módulo de ensino da memória virtual, qual acreditamos que sua proximidade com o módulo de gerencia de memória poderia trazer um melhor aproveitamento de ambos.

Por fim, alcançamos uma ferramenta de aprendizagem concreta e valiosa durante este segundo semestre de 2020, obtivemos valores sólidos e boa parte dos casos de desenvolvimento foram abordados.

Referências

- [1] William Stallings.
”**Operating Systems-Internals and Design Principles**” 7th Edition.
- [2] Maurice Bach. ”**Design of the Unix operating System**”.
<http://www.scielo.org.mx/pdf/cys/v19n2/v19n2a13.pdf>
- [3] open source. ”**Apache Groovy**”.
<https://groovy-lang.org>
- [4] Alfred V. Aho; Jeffrey D. Ullman: ”**Fundamentals of algorithmics**”.
Computer Science Press, Madison Avenue, New York, 1992.
- [5] Pedro Amado. ”**Escalonamento de Processos e Threads**”.
<https://slideplayer.com.br/slide/3263129/>

Memorial

”χαριτωμενο πόνυ και μαγικός καπνός”