Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

Ciência da Computação

Trabalho Prático I

BCC266 - Organização de Computadores

Caio Lucas Pereira da Silva, Vinicius Nunes dos Anjos, Thalles Felipe Rodrigues de Almeida Santos

Professor: Pedro Henquire Lopes Silva

Ouro Preto
15 de janeiro de 2023

Sumário

Lista de Tabelas

Lista de Códigos Fonte

1 Introdução

Para este trabalho é necessário entregar o código em C e um relatório referente ao que foi desenvolvido. O algoritmo a ser desenvolvido é o de uma máquina universal.

As máquinas universais são dispositivos teóricos capazes de simular qualquer processo computacional. Elas foram propostas pela primeira vez por Alan Turing em 1936, como parte de seu trabalho sobre a decidibilidade matemática. Ele mostrou que uma máquina universal, também conhecida como máquina de Turing, poderia ser programada para realizar qualquer tarefa computacional, desde o cálculo simples até a resolução de problemas complexos. Isso levou à criação da teoria da computação, que é hoje uma área fundamental da ciência da computação.

A codificação deve ser feita em C, usando somente a biblioteca padrão da GNU, sem o uso de bibliotecas adicionais. Além disso, deve-se usar um dos padrões: ANSI C 89 ou ANSI C 99.

1.1 Especificações do problema

Este código simula uma máquina universal, que é capaz de simular qualquer processo computacional. O problema específico que ele tenta resolver é a execução de instruções em uma máquina virtual, que é uma representação de uma máquina real. O programa pode ser executado passando dois argumentos na linha de comando: o tipo de instrução (random ou file) e o tamanho da RAM (se for random) ou o nome do arquivo de instruções (se for file).

1.2 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code. [1]
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf LaTeX. [2]

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *Live Share*: ferramenta usada para *pair programming* à distância.
- *Valgrind*: ferramentas de análise dinâmica do código.

1.4 Especificações da máquina

A máquina onde o desenvolvimento e os testes foram realizados possui a seguinte configuração:

- Processador: AMD Ryzen 5-5500U.
- Memória RAM: 8GB.
- Sistema Operacional: Linux Pop!_OS.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar o seguinte comando para rodar o arquivo *Makefile* disponível:

Compilando o projeto

```
make
```

Usou-se para a compilação as seguintes opções:

[1] VScode está disponível em https://code.visualstudio.com/
[2] Disponível em https://www.overleaf.com/

- *-o*: para definir o arquivo de saída.

- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.

- *-Wall*: para mostrar todos os possível *warnings* do código.

- *-c*: para compilação do código e geração dos arquivos objetos.

Para a execução do programa basta digitar :

```
./exe random 'tamanhoRam'
```

ou

```
./exe file 'arquivo'
```

Onde o arquivo de entrada contém o tamanho da matriz e seus valores.

2 Desenvolvimento

O desenvolvimento foi realizado utilizando da técnica de pair programming, onde todos os integrantes do grupo programaram e participaram ativamente do código ao mesmo tempo. O uso das ferramentas Live Share para o compartilhamento de código e do Discord para a comunicação em equipe foi o que viabilizou o uso da técnica citada.

2.1 Criando as operações

A primeira função é a de soma, sendo ela usada de base para outras funções futuras. Ela simplesmente recebe dois números inteiros como entrada e retorna a soma deles.

A seguir o código da função:

```
int sum(int value1, int value2)
{
  return value1 + value2;
}
```

Código 1: Função de soma.

A função de subtração é uma função simples que recebe dois números inteiros como entrada e retorna o valor do primeiro pelo segundo subtraído.

A seguir o código da função:

```
int sub(int value1, int value2)
{
  return value1 - value2;
}
```

Código 2: Função de subtração.

A seguir temos as funções derivadas destas duas primeiras, começando pela multiplicação. A função de multiplicação é uma função que recebe dois números inteiros como entrada e retorna o resultado da multiplicação entre eles. Ela inicializa uma variável "result"com o valor 0 e imprime os valores de entrada. Em seguida, ela verifica se algum dos valores de entrada é 0 e, se for, retorna 0 imediatamente.

Em seguida, ela cria duas variáveis "absValue1" e "absValue2" que armazenam os valores absolutos de "value1" e "value2", respectivamente. Isso é feito para garantir que a multiplicação seja realizada com números positivos, independentemente do sinal dos valores de entrada.

Então, ela usa um loop "for" para adicionar "absValue1" a "result" o número de vezes de "absValue2". Isso simula a multiplicação.

Por fim, ela verifica se algum dos valores de entrada é negativo e, se for, inverte o sinal de "result" antes de retorná-lo.

A seguir o código da função:

```
int mult(int value1, int value2)
{
  int result = 0;

  // Se um dos valores for 0, o resultado e 0.
  if (value1 == 0 || value2 == 0)
    return 0;

  // Alterando os sinais se necessario.
  int absValue1 = (value1 < 0) ? -value1 : value1;
  int absValue2 = (value2 < 0) ? -value2 : value2;

  for (int i = 0; i < absValue2; i++)
  {
```

```
    result = sum(result, absValue1);
  }

  // Alterando os sinais do resultado.
  result = ((value1 < 0 && value2 > 0) || (value1 > 0 && value2 < 0)) ? -
      result : result;

  return result;
}
```

Código 3: Função de multiplicação.

A função de divisão é uma função que recebe dois números inteiros como entrada e retorna o resultado da divisão do primeiro pelo segundo. Ela inicializa uma variável "result" com o valor 0.

Em seguida, ela cria duas variáveis "absValue1" e "absValue2" que armazenam os valores absolutos de "value1" e "value2", respectivamente. Isso é feito para garantir que a divisão seja realizada com números positivos, independentemente do sinal dos valores de entrada.

Então, ela usa um loop "while" para subtrair "absValue2" de "absValue1" até que "absValue1" seja menor que "absValue2". A cada iteração do loop, ela adiciona 1 à "result", pois essa é a contagem de vezes que "absValue2"foi subtraído de "absValue1". Isso simula a divisão.

Por fim, ela verifica se algum dos valores de entrada é negativo e, se for, inverte o sinal de "result" antes de retorná-lo.

A seguir o código da função:

```
int division(int value1, int value2)
{
  int result = 0;

  int absValue1 = (value1 < 0) ? -value1 : value1;
  int absValue2 = (value2 < 0) ? -value2 : value2;

  while (absValue1 >= absValue2)
  {
    absValue1 = sub(absValue1, absValue2);
    result = sum(result, 1);
  }

  result = ((value1 < 0 && value2 > 0) || (value1 > 0 && value2 < 0)) ? -
      result : result;

  return result;
}
```

Código 4: Função de divisão.

A função de exponenciação é uma função que recebe dois números inteiros como entrada e retorna o resultado da exponenciação do primeiro número elevado ao segundo. Ela inicializa uma variável "result" com o valor 1.

Então, ela usa um loop "for" para multiplicar "value1" com "result" o número de vezes de "value2". Isso simula a exponenciação.

Por fim, ela retorna "result" como o resultado da exponenciação.

A seguir o código da função:

```
int expo(int value1, int value2)
{
  int result = 1;

  for (int i = 0; i < value2; i++)
```

```
  {
    result = mult(result, value1);
  }

  return result;
}
```

Código 5: Função de exponenciação.

A função Fibonacci é uma função que recebe um inteiro “stop” como entrada e retorna o n-ésimo termo da sequência de Fibonacci. Ela inicializa três variáveis inteiras “primary”, “secondary” e “result” com os valores 0, 1 e 0, respectivamente.

Então, ela usa um loop “for” para calcular o próximo termo da sequência de Fibonacci, adicionando “primary” e “secondary” e armazenando o resultado em “result”. Então, ela atualiza “primary” para “secondary” e “secondary” para “result”. “stop - 2” vezes, já que os dois primeiros termos são 0 e 1.

Por fim, ela retorna “result” como o n-ésimo termo da sequência de Fibonacci.

A seguir o código da função:

```
int fibo(int stop)
{
  int primary = 0;
  int secondary = 1;
  int result = 0;

  for (int i = 0; i < stop - 2; i++)
  {
    result = sum(primary, secondary);
    primary = secondary;
    secondary = result;
  }

  return result;
}
```

Código 6: Função de Fibonacci.

3 Experimetos

Para os experimêntos, foram feitos três testes no modo random e três usando o modo file. Como os valores são aleatórios, o uso de memória e o tempo de execução variam bastante. O hardware usado foi o especificado na introdução.

Além do mais, para saber o tempo de execução e o uso de memória foi utilizado o *valgrind* e o *time* do linux. Cada entrada foi executada três vezes. Vale ressaltar que para cada entrada, o código foi executado duas vezes, uma usando o comando *time* e outra usando o comando *valgrind*. Isso foi feito pois caso os dois fossem executados juntos, o tempo de execução seria afetado.

A saída do código também pode variar muito, mas a título de exemplo, seria algo do tipo:

```
Iniciando a ámquina...
  > RAM          [ 0 ] : 118
                 [ 1 ] : 107
                 [ 2 ] : 137
                 [ 3 ] : 118
                 [ 4 ] : 95
                 [ 5 ] : 73
                 [ 6 ] : 109
                 [ 7 ] : 73
                 [ 8 ] : 80
                 [ 9 ] : 106
  > Exponenciando RAM[9] (106) com RAM[0] (118) e salvando na RAM[6] (0).
  > Dividindo RAM[3] (118) com RAM[0] (118) e salvando na RAM[4] (1).
  > Levando çãinformao (1) para a RAM[6].
  > RAM          [ 0 ] : 118
                 [ 1 ] : 107
                 [ 2 ] : 137
                 [ 3 ] : 118
                 [ 4 ] : 1
                 [ 5 ] : 73
                 [ 6 ] : 1
                 [ 7 ] : 73
                 [ 8 ] : 80
                 [ 9 ] : 106
```

Código 7: Exemplo de saída.

3.1 Modo random

Entrada 1 O primeiro teste foi feito passando o valor de 100 para a RAM com o seguinte comando:

Comando da entrada 1 do modo random

```
$ time ./exe random 100 $ valgring ./exe random 100
```

O tempo de execução do código foi de 0.137 segundos e o uso de memória foi de 1,584 bytes.

Entrada 2 O segundo teste foi feito passando o valor de 500 para a RAM com o seguinte comando:

Comando da entrada 2 do modo random

```
$ time ./exe random 500 $ valgring ./exe random 500
```

O tempo de execução do código foi de 0.145 segundos e o uso de memória foi de 3,184 bytes.

Entrada 3 O terceiro teste foi feito passando o valor de 1000 para a RAM com o seguinte comando:

Comando da entrada 2 do modo random

```
$ time ./exe random 1000 $ valgring ./exe random 1000
```

O tempo de execução do código foi de 0.147 segundos e o uso de memória foi de 5,184 bytes.

3.2 Modo file

Entrada 1 O primeiro teste foi feito passando o seguinte arquivo como entrada:

```
10 3
3 1 2 3
4 4 5 6
5 7 8 9
```

Código 8: Arquivo de entrada 1.

Para executar o código, os seguintes comandos foram rodados:

Comando da entrada 1 do modo file

```
$ time ./exe file ./tests/test1.txt $ valgring ./exe file ./tests/test1.txt
```

O tempo de execução do código foi de 0.131 segundos e o uso de memória foi de 5,680 bytes.

Entrada 2 O segundo teste foi feito passando o seguinte arquivo como entrada:

```
5 5
1 2 3 4
4 3 2 1
3 2 1 0
6 4 2 4
5 3 2 1
```

Código 9: Arquivo de entrada 2.

Para executar o código, os seguintes comandos foram rodados:

Comando da entrada 2 do modo file

```
$ time ./exe file ./tests/test2.txt $ valgring ./exe file ./tests/test2.txt
```

O tempo de execução do código foi de 0.135 segundos e o uso de memória foi de 5,692 bytes.

Entrada 3 O terceiro teste foi feito passando o seguinte arquivo como entrada:

```
20 2
2 19 1 0
4 9 3 4
```

Código 10: Arquivo de entrada 3.

Para executar o código, os seguintes comandos foram rodados:

Comando da entrada 3 do modo file

```
$ time ./exe file ./tests/test3.txt $ valgring ./exe file ./tests/test3.txt
```

O tempo de execução do código foi de 0.129 segundos e o uso de memória foi de 5,704 bytes.

4 Resultados

A seguir está um resumo dos resultados obtidos a partir de cada entrada, bem como os pontos fortes e fracos do código-fonte do trabalho.

4.1 Resumo dos experimentos

Modo random:

Entrada	Tempo de execução (em segundos)	Consumo de memória (em bytes)
1	0.137	1,584
2	0.145	3,184
3	0.147	5,184

Tabela 1: Resumo dos testes da entrada 1.

Modo file:

Teste	Tempo de execução (em segundos)	Consumo de memória (em bytes)
Teste1	0.131	5,680
Teste 2	0.135	5,692
Teste 3	0.129	5,704

Tabela 2: Resumo dos testes da entrada 2.

4.2 Pontos fortes

- O código está bem escrito e indentado, o tornando fácil de ler e entender.
- As operações estão todas separadas em funções.
- O switch não repete código desnecessário.

4.3 Pontos fracos

- O código não lida com números flutuantes.
- Divisão de um número menor por um maior retorna zero.
- Exponenciação pode falhar quando lidando com números muito grandes

5 Considerações Finais

Neste trabalho prático, analisamos uma implementação simples de uma máquina virtual que pode realizar operações aritméticas básicas, como adição, subtração, multiplicação, divisão e exponenciação, bem como uma sequência de fibonacci. O código também inclui funções para inicializar e executar a máquina virtual e interrompê-la. É importante destacar que a máquina universal desenvolvida neste estudo é apenas um exemplo da teoria da computação e ainda há muito espaço para aperfeiçoamento e novas descobertas.

Em resumo, este estudo demonstrou a viabilidade da programação e teste de uma máquina universal, o que é um passo importante na compreensão da capacidade de computação e suas limitações. A máquina universal desenvolvida é apenas um exemplo da teoria da computação, e ainda há muito espaço para aperfeiçoamento e novas descobertas.