PUCRS - Escola Politécnica
Disciplina: Sistemas Operacionais - 2020/1 - Trabalho Prático - FASE 2 + ES
Prof. Fernando Luís Dotti

1. Definição da Máquina Virtual (MV)

Nossa máquina virtual (MV) tem CPU e Memória.

1.1 Memória

Considere a memória como um array contíguo de posições de memória. Cada posição de memória são 4 bytes. A memória tem 1024 posições.

tamMemoria = 1024
array mem[tamMemoria] of posicaoDeMemoria // adotamos 32 bits

Decisão em aula:

1) ESTRUTURA DE ARMAZENAMENTO CADA posicaoDeMemoria É UM OBJETO COM OS ATRIBUTOS NECESSARIOS PARA CODIFICAR:
   OPCODE
   REG 1..8
   REG 1..8
   PARAMETRO K OU A

1.2 CPU - versão 3

O processador possui os seguintes registradores:

- um contador de instruções (PC)
- oito registradores, 0 a 7

O conjunto de instruções é apresentado na tabela a seguir, adaptado de [1].
A tabela atual está revisada para não ter mais operações a nível de bit. As colunas em vermelho substituem a codificação em bits para designar os registradores e parâmetros utilizados, e compõem os "campos" de uma posicaoDeMemoria (acima)

No.	Mnemonic / OPCODE	Description	Syntax	Micro- operation	REG	REG	PAR
J - Type Instructions							
1	JMP	Direct Jump	JMP k	PC ← k			
2	JMPI		JMPI Rs	PC ← Rs	Rs		k
3	JMPIG		JMPIG Rs, Rc	if Rc > 0 then PC ← Rs Else PC ← PC +1	Rs	Rc	
4	JMPIL		JMPIL Rs, Rc	if Rc < 0 then PC ← Rs Else PC ← PC +1	Rs	Rc	
5	JMPIE		JMPIE Rs, Rc	if Rc = 0 then PC ← Rs Else PC ← PC +1	Rs	Rc	
I - Type Instructions							
6	ADDI	Immediate addition	ADDI Rd, k	Rd ← Rd + k	Rd		k
7	SUBI	Immediate subtraction	SUBI Rd, k	Rd ← Rd – k	Rd		k
8	ANDI	Immediate AND	ANDI Rd, k	Rd ←Rd AND k	Rd		k
9	ORI	Immediate OR	ORI Rd, k	Rd ←Rd OR k	Rd		k
10	LDI	Load immediate	LDI Rd, k	Rd ← k	Rd		k
11	LDD	Load direct from data memory	LDD Rd,[A]	Rd ← [A]	Rd		A
12	STD	Store direct to data memory	STD [A],Rs	[A] ← Rs	Rs		A
R2 – Type Instructions							
13	ADD	Addition	ADD Rd, Rs	Rd ← Rd + Rs	Rd	Rs	
14	SUB	Subtraction	SUB Rd, Rs	Rd ← Rd - Rs	Rd	Rs	
15	MULT	multiplication	MULT Rd, Rs	Rd ← Rd * Rs	Rd	Rs	
16	LDX	Indirect load from memory	LDX Rd,[Rs]	Rd ← [Rs]	Rd	Rs	
17	STX	Indirect storage to memory	STX [Rd],Rs	[Rd] ←Rs	Rd	Rs	
R1 – Type Instructions							
18	SWAP	Swap nibbles	SWAP	Rd7←Rd3, Rd6←Rd2, Rd5←Rd1, Rd4←Rd0	Rd		
19	STOP						
20	TRAP	interrupção de SW	TRAP R1 R1	R1=1 read, R1=2 write, R2=endereço de mem para ou com o conteúdo			

1.3 Programas

Os programas podem ser:
ESCRITOS EM TXT
OU
CODIFICADO EM JAVA COMO A CRIACAO DE UM VETOR DE OBJETOS "posicaoDeMemoria"

Construa os seguintes programas para a nossa MV, no mínimo os que seguem:

a) P1: o programa escreve em posições sabidas de memórias os 10 números da sequência de fibonacci. Ou seja, ao final do programa a memória tem estes 10 números em posições convencionadas no programa.
b) P2: o programa le um valor de uma determinada posição (carregada no inicio),
   se o número for menor que zero coloca -1 no início da posição de memória para saída;
   se for maior que zero este é o número de valores

da sequencia de fibonacci a serem escritos em sequencia a partir de uma posição de memoria;

c) P3: dado um inteiro em alguma posição de memória,
se for negativo armazena -1 na saída;
se for positivo responde o fatorial do número na saída.

d) P4: para um N definido (5 por exemplo)
o programa ordena um vetor de N números em alguma posição de memória;
ordena usando bubble sort
loop ate que não swap nada
passando pelos N valores
faz swap de vizinhos se da esquerda maior que da direita

Possível solução para P1:

```
0   LDI  R1, 0          // 1ro valor
1   STD[50], R1         // de 50 a 60 estarao os numeros de fibonacci
2   LDI R2, 1           // 2o valor da sequencia
3   STD[51], R2
4   LDI R8, 52          // proximo endereco a armazenar proximo numero
5   LDI R6, 6           // 6   é proxima posição de mem (para pular para ela depois)
6   LDI R7, 61          // final
7   LDI R3, 0
8   ADD R3, R4          // R3 =+R4
9   LDI R1,0
10  ADD R1, R2
11  ADD R2, R3
12  STX R8, R3
13  ADD R8, 1
14  SUB R7, R8
15  JMPIG R6, R7
16  STOP
17
…
50  ….                  // resultado será armazenado aqui - vide código
51
…
60  …
```

```
0   LDI  R1, 0          // 1ro valor
1   STD[50], R1         // de 50 a 60 estarao os numeros de fibonacci
2   LDI R2, 1           // 2o valor da sequencia
3   STD[51], R2
4   LDI R8, 52          // proximo endereco a armazenar proximo numero
5   LDI R6, 6           // 6   é proxima posição de mem (para pular para ela depois)
6   LDI R7, 61          // final
7   LDI R3, 0
8   ADD R3, R1          // R3 =+R2
9   LDI R1,0
10  ADD R1, R2
11  ADD R2, R3
12  STX R8, R2
13  ADD R8, 1
14  SUB R7, R8
15  JMPIG R6, R7
16  STOP
17
…
50  DATA  0
51  DATA  1
52  DATA  1
53  DATA  2
54  DATA  3
55  DATA  5
56  DATA  8
57  DATA  13
58  DATA  21
59  DATA  34
60  DATA  55
…
```

```
R1 = 0/ 0/ 1/ 0/ 1/ 0/2 0/ 3 …
R2 = 1/ 1/ 2/ 3 /5  …
R3 = 0/0/ 0/1 /0/1 0/2  …
R4
R5
R6 = 6
R7 = 61/ 8/ 61/ 7/ 61/ 6/ 61 …
R8 = 52/ 53/ 54 …
```

1.4 A CPU em Java (ou outra linguagem)

A CPU É UM GRANDE LOOP,
EM CADA ITERACAO
CARREGA-SE A INSTRUCAO APONTADA POR PC
(A PALAVRA NA POSICAO DA MEMORIA INDEXADA POR PC)
EXECUTA-SE A INSTRUCAO CONFORME SEUS DADOS,
MODIFICANDO REGISTRADORES E POSICOES DE MEMORIA
QUANDO ENCONTRAR STOP O PROGRAMA PÁRA.

A OPERACAO DE STORE EM UMA POSICAO DE MEMORIA ADOTA A DEFINICAO QUE ARMAZENA O VALOR EM “PARAMETRO" DO OBJETO “posicaoDeMemoria” QUE ESTA NAQUELA POSICAO DA MEMORIA.
ADOTA-SE UM OPCODE NOVO, CHAMADO DE “DADO”, QUE INDICA QUE NAQUELA POSICAO TEMOS UM DADO E NAO UMA INSTRUCAO.

1.5 Carregador

UM CARREGADOR, QUANDO INVOCADO TENDO COMO PARAMETRO UM PROGRAMA, LÊ O PROGRAMA SEQUENCIALMENTE E CARREGA A PARTIR DA POSICAO 0 DA MEMÓRIA

1.6 Shell

UM SHELL É UM PROGRAMA QUE ESPERA UMA ENTRADA DO USUARIO
(A ENTRADA É UM NOME DE PROGRAMA)
INVOCA O CARREGADOR, PASSANDO O NOME DO PROGRAMA COMO PARAMETRO
E LIBERA A MAQUINA VIRTUAL (ITEM 4) PARA EXECUTAR A PARTIR DA POSICAO 0

2. UM SISTEMA OPERACIONAL MULTIPROGRAMADO PARA A VM (acima)

Agora vamos construir um sistema capaz de executar vários programas simultaneamente.
Para isso precisamos de um Gerente de Memória, capaz de alocar e deslocar memória para programas, dinamicamente.
Também necessitaremos a representação de vários processos e seu escalonamento, com um Gerente de Processos.

2.1 Gerente de Memória - Fase 1

Vamos dividir a memória em "partições fixas”, o esquema mais simples para suportar multiprogramação.
Estude este mecanismo. Projete as estruturas necessárias para gerenciar a memória segundo este esquema.
Implemente para a nossa VM.

2.2 Gerente de Processos - Fase 1

Para escalonar processos vamos implementar um sistema de time-slice. Significa que cada processo escalonado usa a cpu por um tempo fixo. Os processos formam uma fila circular e usam a CPU conforme esta fila.

2.3 Extensões da CPU para Fase 1

Suporte a relocação

Endereços lógicos traduzidos para endereços físicos:
A cada acesso, traduz:
EndFisico = EndLogico + offSetPartição

Suporte a proteção de memória

Processo pode somente acessar sua partição.
Antes de realizar cada acesso, avalia se
EndInicioPartição <= EndFisico < EndFimPartição

Registradores adicionais: base e limite (de memória)
Função adicional (em hw): tradução de endereço

2.4 Entrada e Saída

Nosso sistema operacional so faz leituras do teclado e escritas na tela, de valores inteiros.
Assim, cada operação só envolve um dispositivo (e não precisamos representar múltiplos).

Interface para chamadas de sistema

Nosso sistema operacional oferece duas chamadas para entrada e saída, read e write.
Convencionamos que: o número da chamada de sistema é armazenado em R1
e os parâmetros conforme especificação da rotina:

Read: R1 = 1, R2: endereço de escrita na memória do valor lido, ou seja, uma chamada seria

```
LDI R1,1
LDI R2, endereço destino da leitura
Trap
```

Write: R1 = 2, R2: endereço do valor a ser escrito

```
LDI R1, 2
LDI R2, endereço do valor a ser escrito
Trap
```

Trap : na CPU significa disparar rotina para tratar

Chamadas de sistema

A interrupção de Software, ou Trap, salva o contexto do processo em execução, inicia a execução da rotina de tratamento da Trap.

Se for rotina de leitura, enfileira um pedido para a Console para ler um valor.
Se for rotina de escrita, enfileira um pedido para a Console para escrever um valor.
Em ambos os casos, coloca o processo no estado bloqueado, esperando sinalização de fim da operação, e
Escalona outro processo segundo a rotina do escalonador.

Console

A console é um processo concorrente (thread) que pega as requisições da fila, executa, e gera uma interrupção para indicar que a operação está pronta.

O pedido de leitura lê um valor inteiro da console. O usuário fornece o valor.
O pedido de escrita imprime na tela o valor escrito.

Para facilitar, em cada execução de entrada ou saída, o sistema vai imprimir qual processo e que operação está sendo feita.

Uma vez que a operação seja realizada:

- no caso de escrita, interrompe a CPU dizendo de qual processo a operação acabou
- no caso de leitura, escreve na posição de memória dada como parâmetro e interrompe a CPU dizendo de qual processo a operação está pronta

Rotina de tratamento de interrupção - I/O pronto

A rotina de tratamento desta interrupção deve transferir o processo do estado bloqueado para o estado pronto, e retomar a execução de processos, conforme a política do SO.