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

*Trabalho Prático. **Terceira** parte. Em grupos de até 4 alunos.*

Sobre a Máquina Virtual + Gerente de Memória (já entregues), construir gerente de processos, escalonamento e IO.
A entrega é em data conforme calendário da disciplina.

O ANEXO contém as definições de Máquina Virtual e do Gerente de Memória.

3. Gerente de Processos (GP)

O Gerente de Processos é responsável pela:

- criação de processos;
- escalonamento, mudanças de estados;
- terminação de processos.

3.1. Criação de Processos

Dado um programa a ser executado, solicitado pelo usuário através de um Shell, gera-se uma solicitação de criação de processo ao gerente de processos, com o nome (identificador) do programa.
O gerente de processos é responsável pela criação do novo processo, envolvendo:

- Aloca um novo identificador ao processo;
- Aloca um Process Control Block (estrutura de dados) para representar o processo;
- Solicita frames de memória para conter todo processo em memória (assume-se que o tamanho é conhecido), incluindo área de código e dados;
- Carrega as páginas do programa nos frames;
- Armazena a tabela de páginas no PCB;
  (até aqui foi ja feito na entrega anterior, agora voce vai organizar isso como uma funcionalidade do GP, e **atenção** aos passos seguintes, pois eles têm novidades)
- Salvamento do estado da CPU: Nesta versão do sistema, os processos poderão ser escalonados. Isto significa que eles vão poder utilizar a CPU diversas vezes na sua vida. Então o SO deve salvar o contexto de execução de um processo saindo da CPU, assim como restaurar o estado de um processo quando colocar ele na CPU (dispatch). Assim, o Process Control Block deve ter atributos para permitir salvar o estado de execução do processo na CPU. Isto inclui todos os registradores, PC, IR. No início, PC=0.
- Depois de criar todas estruturas acima, o processo é colocado na fila de prontos.
- Se não á processo executando, o Dispatcher é liberado para escolher um entre os processos da fila de prontos e executar.

3.2. Finalização de Processos

Se um processo deve ser finalizado (operação STOP, erro de execução, violação de endereçamento, etc), uma rotina de finalização de processo, do GP, é invocada. Esta rotina é responsável por deslocar todas estruturas alocadas por este processo. Desaloca frames, PCB, etc.
Como no nosso sistema um processo só termina quando está executando (não existe opção de um processo matar outro), ao final desta rotina de tratamento, o Dispatcher é liberado para escolher um processo para executar na CPU.

4. Escalonamento

O escalonamento é uma tarefa de gerência de processos. Entretanto separamos aqui para melhor explanação.
Consideramos o Processos responsável por criar e finalizar processos apenas. E um escalonador para fazer o “dispatching”.
O Dispatcher continuamente escolhe um entre os processos na fila de prontos (ready) para executar na CPU.
Ele seta na CPU o contexto de execução do processo e libera a CPU para execução do processo escolhido a partir do estado onde estava (PC e demais registradores …). O Dispatcher aguarda nova liberação para execução.

4.1. Dispatcher RR (Round-Robin)

Implementa a política Round-Robin. Depois de uma fatia de tempo, o processo utilizando a CPU retorna para a fila de prontos e outro processo é escolhido pelo dispatcher para executar.

Para **simular** a fatia de tempo, configuraremos um determinado numero X de instruções na CPU que o processo eXecuta. por exemplo X=5, como sendo o tempo de permanência de um processo na CPU. Ao completar X instruções, gera-se um sinal de interrupção significando final da fatia de tempo, e a CPU desvia para uma rotina específica para isso. Num sistema completo, esta interrupção é (seria) gerada por um relógio do sistema.

Rotina de tratamento de final de fatia de tempo.
salva o estado do processo que estava rodando
retorna este processo para o final da fila de prontos
libera o Dispatcher para escolher o próximo processo a executar.

4.2. Processo Rodando

O sistema operacional deve ter uma variável “processoRodando" que aponta para o PCB do processo executando neste momento.

4.3. Fila de Processos Prontos

O sistema operacional tem uma fila de processos prontos que trabalha como uma FIFO.

5. Entrada e Saída

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

5.1. 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:

5.1.1 Leitura

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

```
LDI R1,1       // leitura
LDI R2, 30     // escrever o valor lido na posição 30 da memória
Trap
```

5.1.2 Escrita

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

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

5.1.3 Trap - NOVA INSTRUCAO DA CPU

Trap : interrupção de SW. 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 e provoca o escalonamento de outro processo.

5.2. 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 leitura, escreve na posição de memória dada como parâmetro
* no caso de leitura ou escrita, interrompe a CPU dizendo de qual processo a operação acabou

5.3. 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.

ANEXO

1.Definição da Máquina Virtual (MV)

Nossa máquina virtual (MV) tem CPU e Memória.

1.1 CPU

O processador possui os seguintes registradores:

- um contador de instruções (PC)
- Um registrados de instruções (IR)
- 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 vide 1.2.

No.	**Mnemonic / OPCODE**	Description	Syntax	Micro- operation	**R1**	**R2**	**P**
J - Type Instructions							
1	JMP	Direct Jump	JMP k	PC ← k			K
2	JMPI		JMPI Rs	PC ← Rs	Rs		
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	
6	JMPIM	Adicionada dia 25 ago	JMPIM [A]	PC ← [A]			A
7	JMPIGM	Adicionada dia 25 ago	JMPIGM [A], Rc	if Rc > 0 then PC ← [A] Else PC ← PC +1		Rc	A
8	JMPILM	Adicionada dia 25 ago	JMPILM [A], Rc	if Rc < 0 then PC ← [A] Else PC ← PC +1		Rc	A
9	JMPIEM	Adicionada dia 25 ago	JMPIEM [A], Rc	if Rc = 0 then PC ← [A] Else PC ← PC +1		Rc	A
I - Type Instructions							
10	ADDI	Immediate addition	ADDI Rd, k	Rd ← Rd + k	Rd		k
11	SUBI	Immediate subtraction	SUBI Rd, k	Rd ← Rd – k	Rd		k
12	LDI	Load immediate	LDI Rd, k	Rd ← k	Rd		k
13	LDD	Load direct from data memory	LDD Rd,[A]	Rd ← [A]	Rd		A
14	STD	Store direct to data memory	STD [A],Rs	[A] ← Rs	Rs		A
R2 – Type Instructions							
15	ADD	Addition	ADD Rd, Rs	Rd ← Rd + Rs	Rd	Rs	
16	SUB	Subtraction	SUB Rd, Rs	Rd ← Rd - Rs	Rd	Rs	
17	MULT	multiplication	MULT Rd, Rs	Rd ← Rd * Rs	Rd	Rs	
18	LDX	Indirect load from memory	LDX Rd,[Rs]	Rd ← [Rs]	Rd	Rs	
19	STX	Indirect storage to memory	STX [Rd],Rs	[Rd] ←Rs	Rd	Rs	
R1 – Type Instructions							
20	SWAP	SWAP regs	SWAP Ra, Rb	T ← Ra Ra ← Rb Rb ←T			
21	STOP						

1.2 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
```

Cada posiçãoDeMemória codifica **[** OPCODE, 1 REG de 1..8, 1 REG de 1..8, PARAMETRO K OU A conforme operação **]**. Em um sistema real estes dados são codificados em bits de uma palavra. No nosso trabalho, podemos adotar que uma posicaoDeMemoria é um registro (objeto) com estes atributos. Note que no caso da posição de memória não ter uma instrução, temos que adotar então uma codificação própria. Neste caso podemos ter um OPCODE especial para significar uma posição de dados, e, no campo de K ou A, temos um valor inteiro. Um valor inteiro é suficiente pois a nossa arquitetura manipulará inteiros apenas.

1.3 Funcionamento da CPU: o ciclo de instruções

A CPU executa o ciclo de instruções. Dado um valor no PC, ela faz:

```
Loop
    busca a posição de memória apontada por PC, carrega no RI
    Decodifica o RI
        se RI inválido - sinaliza interrupção, acaba esta instrução
        executa operação
                Se erro: sinaliza interrupcao, acaba esta instrução
                conforme operação efetua a mesma e atualiza PC
    se existe interrupção
        desvia para rotina de tratamento
fimLoop
```

Um vetor de interrupções associa o código da interrupção com a rotina a ser executada.

1.4 Programas

Neste momento não temos um Sistema Operacional. Para fazer a Máquina Virtual funcionar, você deve carregar um programa no início da memória, atribuir um valor ao PC (o início do código do seu programa), e liberar a CPU para executar. A CPU vai executar até parar, encontrar um erro, ou então vai entrar em loop conforme o programa estiver correto ou não.

Você deve criar formas de acompanhar esta evolução.

Nossos programas podem ser escritos em TXT e lidos para a memória, ou então eles podem ser codificados em Java como a criação de um vetor de posicaoDeMemoria inicializado em cada posição do vetor como uma linha do programa.

A seguir o programa P1, que escreve em posições sabidas de memória os 10 números da sequência de Fibonacci. Ou seja, ao final do programa a memória tem estes 10 números em posições convencionadas no programa. Avalie se P1 está correto.

Como parte do exercício, construa P2, um programa que 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;
```

Outros programas

```
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

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 (página a seguir):

P1: Verifique se funciona!

```
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 =+R1
9   LDI R1,0
10  ADD R1, R2
11  ADD R2, R3
12  STX R8, R2
13  ADDI 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
…
```

2. Gerente de Memória

2.1 Valores Básicos

O gerente de memória para a VM implementa paginação, com:

Tamanho de Página = 16 palavras (ou posições de memória)

A memória tem 1024 palavras. Assim teremos:

Número de frames = 64

2.2 Fucionalidades do Gerente

Alocação: Dada uma demanda em número de palavras, o gerente deve responder se a alocação é possível e, caso seja, retornar o conjunto de frames alocados : um array de inteiros com os índices dos frames.

Desalocação: Dado um array de inteiros com as páginas de um processo, o gerente desloca as páginas.

Interface - solicita-se a definição clara de uma interface para o gerente de memória.

```
GM {
        Boolean   aloca(IN int nroPalavras,     OUT pags []int)
        Void desaloca(IN  pass []int)
}
```

Estruturas internas: controle de páginas alocadas e disponíveis.

```
tamMemoria = 1024                                   // fornecido anteriormente
array mem[tamMemoria] of posicaoDeMemoria

tamPag = tamFrame = 16
nroFrames = tamMemoria / tamPag    // 64 aqui
```

array frameLivre[nroFrames] of boolean // inicialmente true para todos frames

Os frames terão índices. Cada frame com índice f inicia em (f)*tamFrame e termina em (f+1)*tamFrame -1
Exemplo:

frame	início	fim
0	0	15
1	16	31
2	32	47
3	48	63
4	64 …	
	…	
	…	
62	992	1007
63	1008	1023

2.3. Aspectos do Funcionamento com Paginação

Alocação de Programa:

necessita-se saber o tamanho do programa.
Gerente de memória aloca frames para conter o mesmo.
O retorno é um array de índices de frames.

A partir disso, o programa é copiado na memória de forma a preencher os frames sequencialmente, na ordem indicada no array de alocação.
Desta forma, o array de alocação de frames é também a tabela de páginas do processo.

Exemplo: considere um programa de 5 páginas, que teve alocados os seguintes frames, em ordem
[2, 3, 8, 12, 10]
Chamemos agora este mesmo array de tabela de páginas do processo:
tabPaginas = [2, 3, 8, 12, 10]
assim, tabPaginas[0]=2
tabPaginas[1]=3
…
tabPaginas[4]=10
seja p uma página do processo, tabPaginas[p] é o frame onde a página está alocada

Execução do Processo:

um processo tem uma tabela de páginas como parte do seu estado.
Para executá-lo, a tabela é conhecida. Esta tabela é carregada em uma estrutura do processador para tal.

Cada acesso à memória passa pela translação:

seja A um endereço,
A div tamPag = p é o índice da página que o programa acessa
A mod tamPag = offset é o deslocamento dentro da página
toma-se p e obtém-se
tabPaginas[p] o frame onde a página se encontra
tabPaginas[p] * tamFrame é o início do frame a ser acessado
assim,
(tabPaginas[p] * tamFrame) + offset é o endereço a ser acessado na memória física
ou
(tabPaginas [(A div tamPag)] * tamFrame) + (A mod tamPag)

Pode-se montar uma função de tradução de endereço T que, dado A endereço lógico do programa

T(A) = (tabPaginas [(A div tamPag)] * tamFrame) + (A mod tamPag)

Traduz A para o endereço que deve ser acessado no array.

No processo de tradução deve ser verificado se a página resultante é válida, gerando erro ou permitindo o acesso. A página é válida se o programa tem o índice p da página obtida em A div tamPag = p.

2.4. Testes

Deve-se demonstrar que o sistema pode carregar vários processos em memória. E depois executar cada um sequencialmente, do início ao fim. Deve-se provocar que processos utilizem frames não vizinhos na memória para testar adequadamente a carga e o endereçamento dos processos.