Simulador de Sistema Operativo com Modelo de 5 Estados com threads e Gestão de Memória por segmentação

Sistemas Operativos – Enunciado do Trabalho 2.

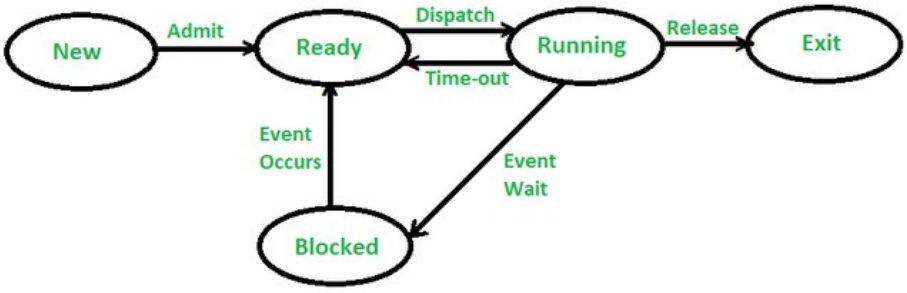
Luís Rato, 2022
Departamento de Informática
Universidade de Évora

Resumo
Este trabalho vem na sequência do trabalho 1. Neste trabalho pretende-se implementar um simulador de sistema operativo mais realista do que no trabalho anterior. Este simulador incluirá a simulação de uma memória linear; a execução de instruções com operações simples com variáveis (cópia, atribuição, soma, multiplicação,...); instruções de controlo (IF e JUMPs); operação de Input/Output; e uso de threads.

O trabalho pode ser realizado individualmente ou em grupos de dois ou três alunos. No Moodle deverá submeter um .zip com os números de aluno no nome do ficheiro, ex "14444_22222.zip" e deverá conter o código fonte do trabalho assim como um relatório em PDF.
Os trabalhos serão testados num sistema Linux Ubuntu, para garantir a máxima compatibilidade aconselha-se a usar um Ubuntu 20.04LTS, ao qual apenas seja adicionado o compilador gcc pelo que deverá instalar essa versão, ou instalar uma máquina virtual com virtualbox ou Vmware de modo a garantir que esteja a usar o ambiente correto. Se for necessário a compilação de vários ficheiros deverá ser incluída, ou uma makefile ou um shell script para a compilação de cada executável com as opções que utilizou.

Modelo de estados

De modo semelhante ao trabalho anterior o modelo de estados a usar será o modelo de 5 estados, com os estados NEW, READY, RUN, BLOCKED, EXIT.



Tipo de programas e processos

O sistema a simular terá um modelo muito simples de programação, e será composto por:

- Um conjunto de instruções simples do processo principal (os detalhes da execução serão descritos mais à frente).
- Variáveis do programa que deverão ser tratadas como variáveis globais, e serão portanto partilhadas entre *threads*, caso existam.
- Instruções da *thread* (um, e apenas um bloco de código)
- Variáveis da *thread*, que deverão ser tratadas como variáveis locais, e portanto não deverão ser partilhadas entre *threads*.

Definição dos programas
Os programas serão definidos num ficheiro de texto com a seguinte estrutura:

```
LOAD t1     (início do 1º programa no instante t1)
INST X
...
INST Y
```

```
THRD Z
INST X
...
INST Y
ENDP Z      (fim do 1º programa)
LOAD t2     (início do 2º programa no instante t2)
INST X
...
INST Y
THRD Z
INST X
...
INST Y
ENDP Z     (fim do 2º programa)
LOAD t3    (início do 3º programa no instante t3)
INST X
INST Y
...
INST X
INST Y
etc...
```

Deste modo, as linhas com os comandos **LOAD t1, THRD Z, ENDP Z** serão os marcadores para os blocos de instruções do programa principal (de LOAD até THRD), e das intruções da thread (de THRD até ENDP). Caso não se queira usar threads o bloco com as instruções da thread poderá ficar vazio:

```
LOAD t1
INST X
...
INST Y
THRD Z
ENDP Z
```

Para facilitar a leitura do input (ficheiro de texto) pode assumir, se quiser, que em cada linha tem sempre uma sequência de quatro caracteres que representa um comando ou uma instrução, seguida de um número inteiro (ainda que nalgumas situações o número inteiro não tenha informação relevante e venha a ser descartado).

Gestão de memória

A memória do sistema será representada por um array de inteiros de dimensão limitada, **int mem[]** com a dimensão de 200 O programa é guardado em memória, representada por um array.

Cada processo deverá gerir o espaço para: guardar as **instruções**; guardar as **variáveis;** e eventualmente para guardar as **instruções da thread** e as **variáveis da/das threads**. Cada processo ou thread tem que reservar espaço para as suas variáveis (inteiras). No mínimo é obrigado a ter espaço para uma variável, o var_0 (“var_zero”) mas pode ter mais. É necessário verificar qual o endereço “x” de dados (Var_x) mais elevado e reservar espaço para todas essas variáveis. Por exemplo se num programa o var_x mais

elevado é 13 (var_13), logo é preciso reservar espaço para 14 variáveis (de var_0 a var_13).

Todas as variáveis na gama de Var_0 a Var_9 são consideradas variáveis locais (não serão partilhadas entre threads), enquanto as variáveis na gama de Var_10 até ao máximo, e.g. Var_100, são variáveis globais (serão partilhadas entre o processo principal e threads, caso existam.)

Naturalmente este é um processo muito simplificado de atribuição de espaço de memória para as variáveis de programa e threads, e que é pouco eficiente, (pois será reservado espaço para variáveis mesmo que não venhama ser usadas) um compilador faria algo semelhante mas de modo muito mais eficiente, reservando apenas o espaço de memória necessário para as variáveis efetivamente usadas no programa. No entanto, é este o método que deve ser seguido neste trabalho.

A memória deverá ser gerida através de segmentação. Deste modo deverá ter os seguintes segmentos para cada processo:

- instruções do programa (1 segmento)
- variáveis do programa (1 segmento)
- instruções da thread (1 segmento)
- variáveis da/das threads (1 ou mais segmentos)

Poderá então incluir no processo (numa estrutura PCB/process control block) variáveis para gerir estes segmentos, por exemplo:

base_cod_prog – início do segmento com as instruções do programa
lim_cod_prog – dimensão do segmento as instruções do programa

base_var_prog – início do segmento com as variáveis do programa
lim_var_prog – dimensão do segmento as variáveis do programa

base_cod_thread – início do segmento com as instruções da thread
lim_cod_thread – dimensão do segmento as instruções da thread

base_var_thread[] – início do segmento com as variáveis das threads
lim_var_thread[] – dimensão do segmento as variáveis das threads
(sugere-se o uso de um array para os segmentos das variáveis das threads uma vez que poderão ser lançadas várias threads, mas pode arbitrar uma dimensão máxima e.g. 4 deste modo cada processo não poderá lançar mais do que 4 threads)

Além destes o Processo deverá incluir naturalmente, variáveis para:

- **Program Counter** do processo (em rigor será o program counter da thread do processo principal)
- **Program counters** das threads, caso existam.
- **Estado da thread** to processo principal.
- **Estado das threads**, caso existam.
- **IDs do processo principal e das threads.**

Usualmente estas informações estão guardadas numa estrutura de nome TCB (Thread Control Block, ou Task Control Block). Note que como o nosso simulador de SO reconhece

a existência de Threads será essencialmente esta informação (do TCB) que deverá circular entre as filas e estados do sistema.

A alocação de memória deverá ser feita aplicando o algoritmo **NEXT FIT.**

Transição entre estados

Neste sistema considere que:

- todas a instruções consomem exatamente 1 instante de tempo de CPU;
- cada chamada de I/O (que implicará a passagem do processo/thread para o estado blocked) demora exatamente 4 instantes de tempo para ser despachada; este tempo só conta para o processo/thread que está na cabeça da fila blocked.
- o escalonamento de processos é feito de acordo com o algoritmo preemptivo Round-Robin. O quantum default deve ser igual a 3 instantes de tempo, mas deve ser configurável.
- quando no mesmo instante, um processo vindo de NEW, um processo/thread de BLOCKED, e *ou do RUN querem transitar para o estado READY, o* processo/thread vindo do BLOCKED tem prioridade, seguido do de RUN, e por fim o processo/os de NEW;
- No estado NEW, cada processo consome exatamente 1 instante de tempo.
- No estado EXIT, cada processo consome exatamente 1 instante de tempo, após o que é apagado do sistema.
- A execução da instrução HLT (halt) termina o processo, passando assim para estado EXIT. Quaisquer threads que existam associadas ao processo devem ser removidas (mesmo que não tenham terminado).
- Quando se tenta criar um processo novo a partir do input usando o "LOAD t" será necessário alocar o espaço na memória principal. Caso não seja possível, a criação do processo aborta e há uma mensagem de erro para o standard ouput: "impossível criar processo número X, o espaço disponível em memória é insuficiente"
- Quando se tenta criar uma thread nova (a partir da instrução NWTH (new thread)) será necessário alocar o espaço para as variáveis da thread na memória principal. Caso não seja possível, a criação da thread aborta e há uma mensagem de erro para o standard ouput: "impossível criar thread número X do processo Y, o espaço disponível em memória é insuficiente"
- Deve assumir um modelos de 3 estados para as threads, ou seja, quando são criadas as threads vão imediatamente para READY (sem passar por NEW), e quando são terminadas e saiem de RUN são imediatamente removidas sem passar por EXIT.

Estrutura do simulador

Para que as prioridades nas transições entre estados sejam cumpridas é essencial percorrer os processos estado-a-estado. Estas transições entre estados devem ser processadas usando funções específicas. Deste modo não só a legibilidade do programa será melhorada como será mais fácil fazer debugging, caso seja necessário.

Considere a seguinte sequência, com as transições entre estados, a execução das instruções, e a atualização do instante de tempo, a qual repetir-se-á enquanto existirem processos por terminar no sistema, ou até se atingir um número máximo de instantes de tempo (e.g. tmax=100).

Transições de estado

- `Blocked2ready()` - verifica a transição de estado de BLOCKED para READY
- `Run2exit_blocked_run()` - verifica a transição de estado de RUN para outros estados (ou seja verifica que a instrução é HALT, ou se a instrução é DISK, ou se já terminou o quantum do Round Robin)
- `New2ready()` - verifica se há processos em NEW e passa-os para READY
- `Ready2run()` - verifica se o processador está livre, e passa um processo/thread de READY para RUN (se a fila de READY não estiver vazia)
- `NewProcess()` - verifica se o atual instante é o instante inicial de algum processo (indicado por "LOAD X" no ficheiro de entrada); cria o processo e coloca-o em NEW (o processo deverá reservar o espaço em memória necessário)
- `RemoveProcess()` - verifica se existem processos em EXIT, e remove-os (todas as zonas de memória que estivessem alocadas deverão passar a estar livres)

Executa instrução

- `Execute()` - executa a instrução do processo no estado RUN (ADDX, MULX, ZERO, etc, ...)

Atualiza instante

- Atualiza o instante de tempo: `t=t+1;`

Instruções

As instruções que constituem os programas são definidas do seguinte modo:

Instrução	Var.	Descrição	Código
ZERO	X	Var_0 = X	0
COPY	X	Copia var_0 para var_X (var_x = var0)	1
DECR	X	Decrementa var_X	2
NWTH	X	Cria uma Thread número X. O parâmetro de entrada da thread está na VAR_X do processo principal, e será copiado para VAR0 da thread (var. Local da thread), o número da thread X será copiado para VAR1 (var. Local da thread)	3
JFRW	X	Jump forward X instruções	4
JBCK	X	Jump back X instruções	5
DISK	X	Pedido de I/O (processo passa a Blocked)	6
JIFZ	X	Jump if X = zero (se X= zero então salta duas instruções, senão segue para a próxima instrução)	7
PRNT	X	Print/ imprime o valor de Var_X	8
JOIN	X	Espera pelo fim da thread X	9
ADDX	X	Soma Var_0 com Var_x e o resultado vai para Var_0	10
MULX	X	Multiplica Var_0 com Var_x e o resultado vai para Var_0	11
RETN	X	Termina a thread e retorna ao processo principal	12
HALT	X	Halt / termina o processo	20 ou qualquer outro valor

Os programas são definidos, através dum ficheiro de entrada *input.txt.*

Segurança e isolamento entre processos

Caso um processo tente executar uma instrução fora do espaço alocado para o código, o processo é terminado e há uma mensagem de erro “erro de segmentação do processo X” para o standard ouput.

Codificação das instruções do programa

Cada instrução é codificada por dois números inteiros: o primeiro é o código da instrução e o segundo é um parâmetro da instrução (de acordo com a tabela acima, que descreve as instruções).
Nalgumas instruções como DISK e HALT o valor numérico que se segue é irrelevante para a execução, no entanto deverá existir sempre um valor de modo a manter a uniformidade do formato das instruções. Deste modo **cada instrução ocupa sempre dois valores na memória** (duas variáveis inteiras).

Output do simulador

O output para o stdout deve ter o seguinte formato mostrando os processos em cada fila, em cada instante. Os prints e mensagens de erro (segmentação ou memória insuficiente) deverão ser organizados do seguinte modo:

```
...
  T    NEW | READY         | RUN  | BLOCKED      | EXIT
...
09         | P2 P3 TH1 P4  | P5   | P6           |
10         | P2 P3         | P4   | P5 P6        |
11         | P2 P3 TH1     | P4   | P5 P6        |
12         | P2 P3         | TH1  | P5 P6        | P4
> impossível criar processo número 7, o espaço disponível em
memória é insuficiente
13         | P2 P3         | TH1  | P5 P6        |
14         | P6 P2         | P3   | P5           |
> impossível criar thread número 2 do processo 3, o espaço
disponível em memória é insuficiente
14         | P6 P2         | P3   |  P5          |
> print 25
15         | P5 P6         | P2   |              | P5
etc ...
```

Exemplo#1

```
LOAD 0     (início do 1º programa no instante 0)
ZERO 3     (var0 = 3)
COPY 1     (var1 = var0) (ou seja v1=3)
ZERO 4     (var0 = 4)
COPY 2     (var2 = var0) (ou seja v2=4)
NWTH 1     (lança thread 1)
NWTH 2     (lança thread 2)
JOIN 1     (espera pelo fim da thread 1)
JOIN 2     (espera pelo fim da thread 2)
```

```
ZERO 0     (var0 = 0)
ADDX 11    (var0 = var0 + var11) (ou seja v0=v11)
ADDX 12    (var0 = var0 + var12) (ou seja v0=v11+v12)
PRNT 0     (print var0)
HALT 0     (termina)
THRD 0     (início da thread do 1º programa)
MULX 0     (var0 = var0*var0) (var0 tem o parâmetro de entrada)
                              (ou seja eleva v0 ao quadrado
                              v0=v0*v0)
DECR 1     (var1 = var1-1)    (var1 tem o número da thread)
JIFZ 1     (salta 2 se var1==0)
COPY 12    (var12 = var0)
RETN 0     (sai da thread)
COPY 11    (var11 = var0)     (ou seja se o número da thread é 1
                              copia o resultado para v11,
                              caso contrário copia para v12)
RETN 0     (sai da thread)
ENDP 0     (fim do 1º programa)
```

No fim deveremos ter o print da soma de 3*3 com 4*4, ou seja 25.

Exemplo#2 um programa que tentar saltar (para a frente) para fora do segmento das instruções

```
LOAD 0     (início do 1º programa no instante 0)
ZERO 3     (var0 = 3)
COPY 1     (var1 = var0)
ZERO 4     (var0 = 4)
COPY 2     (var2 = var0)
JFRW 5     (salta 6 instruções para a frente)
ADDX 11    (var0 = var0 + var11)
HALT 0     (termina)
THRD 0     (início da thread)
ENDP 0     (fim do programa)
```

Exemplo#3 um programa que tentar saltar (para trás) para fora do segmento das instruções

```
LOAD 0     (início do programa no instante 0)
ZERO 3     (var0 = 3)
COPY 1     (var1 = var0)
ZERO 4     (var0 = 4)
COPY 2     (var2 = var0)
JBCK 5     (salta 6 instruções para trás)
ADDX 11    (var0 = var0 + var11)
HALT 0     (termina)
THRD 0     (início da thread)
ENDP 0     (fim do programa)
```

Exemplo#4 dois processos com instruções de I/O

```
LOAD 1     (início do 1º programa no instante 1)
ZERO 3     (var0 = 3)
COPY 10    (var10 = var0)
DISK 0     (instrução de I/O)
COPY 20    (var20 = var0)
JFWR 1     (salta 2 instruções para a frente)
HALT 0     (termina)
JBCK 1     (salta 1 instrução para a trás)
THRD 0     (início da thread)
ENDP 0     (fim do programa)
LOAD 0     (início do 2º programa no instante 0)
ZERO 3     (var0 = 3)
COPY 1     (var1 = var0)
ZERO 4     (var0 = 4)
COPY 2     (var2 = var0)
DISK 0     (instrução de I/O)
HALT 0     (termina)
THRD 0     (início da thread)
ENDP 0     (fim do programa)
```