Teste 2A de Arquitetura de Computadores
8/06/2016 Duração 2h *sem consulta*

Número: _____ *Nome:* _____

1- 1 val

O fatorial de um número natural pode ser definido como: n! = n (n-1)! sendo 1! = 1

Considere a seguinte implementação recursiva em C:

```
int fact( int n ) {
   if ( n==1 ) return 1;
   else return n*fact(n-1);
}
```

a) Considere a chamada `x=fact(3)`. Diga quantas vezes a função fact será chamada (execução de `call fact`), incluindo esta primeira.

3 vezes

b) Nas condições de a), desenhe ao lado a pilha de execução de todas as chamadas até a condição `n==1` é *Verdade* (indique a posição apontada pelo registo ESP). Assumindo que não usa ebp:

3
end ret
2
end ret
1
end ret <- ESP

2- 1 val

Considere que o software pede a leitura de um determinado bloco de um disco magnético (HDD). Enumere os principais fatores que determinam o tempo de acesso a esse bloco e explique cada um.

Tempo de acesso à pista (seek) – tempo para a cabeça de leitura/escrita chegar à pista pretendida

Tempo de rotação – tempo até o sector pretendido chegar à cabeça para começar a ler o bloco

Tempo de leitura – tempo que demora a ler todos os bytes do bloco

(todo o processo pode ter de se repetir se um bloco consistir em mais de um sector)

3- 1 val

Para que os Sistemas de Operação possam suportar múltiplos processos em execução sem que estes interfiram, a arquitectura *hardware* deve suportar alguns mecanismos. Um deles consiste em algumas instruções serem apenas acessíveis ao Sistema de Operação usando o modo supervisor.

a) Justifique a necessidade deste mecanismo para a partilha de periféricos e IO.

As instruções que permitem aceder aos periféricos (p.e. in e out) têm de ser reservadas ao SO. Só o SO poderá assim alterar os periféricos e fazer IO, gerindo a sua partilha. Os processos têm sempre de pedir ao SO para fazer este tipo de operações.

b) Diga se a instrução CLI (desligar interrupções no CPU) necessita de ser uma instrução privilegiada (de modo supervisor) ou não. Justifique.

Sim, necessita de ser privilegiada. Se qualquer processo a poder usar, pode desligar as interrupções e assim comprometer o funcionamento de operações que usem interrupções, como por exemplo, nas operações de IO, assim como impedir a partilha do CPU (time-sharing) ao não deixar ser interrompido.

4- 1,5 val

Numa determinada arquitectura, o CPU tira partido do *pipelining* das várias fases de execução das instruções.

a) Indique em que condições este mecanismo permite uma melhor taxa de execução de instruções.

No pipeline podem estar várias instruções em execução, cada uma numa das várias fases de execução. Desde que possa manter todas as fases do pipeline ocupadas com instruções, pode completar mais instruções por unidade de tempo.

b) Descreva os potenciais problemas que a execução de instruções de salto (*jumps*) pode ter para o desempenho, face ao que pode suceder quando uma delas aparece no fluxo de instruções no pipeline.

Se aparece um jump a próxima instrução a executar pode não ser a seguinte no pipeline. Mais grave no caso de saltos condicionais, pois só depois da sua execução se pode saber qual a instrução que será executada depois desse jump. O pipeline pode ficar assim vazio (parado) à espera de saber qual a instrução a executar. Para minorar esse problema, o CPU pode escolher colocar logo as instruções destino do salto incondicional no pipeline e, para os condicionais, uma das sequências possíveis (usar branch prediction). Se acertar, já tem as próximas instruções no pipeline.

5- 5 val

Admita uma arquitectura de um computador com as seguintes características: endereçamento de 20 bits, um nível de cache composta por uma cache associativa por grupos (*sets*) de 8 linhas, com escritas diferidas (*write-back*). Cada endereço é interpretado do seguinte modo:

- Os 5 bits menos significativos como o deslocamento num bloco (ou linha)
- Os 7 bits seguintes (do bit 5 ao 11) como o número do grupo

a) Como serão interpretados os 8 bits mais significativos (do bit 12 ao 19)?

Chave ou tag

b) Qual é o tamanho em Bytes de um bloco ou linha?

2^5 = 32 Bytes

c) Quantos grupos (*sets*) existem nesta cache?

2^7 = 128 grupos

d) Qual é a capacidade da cache (em KBytes)?

128 grupos * 8 linhas * 32 Bytes = $2^7 * 2^3 * 2^5 = 2^5 * 2^{10}$ = 32 KBytes

e) No acesso ao endereço `1101 0110 0001 0000 1011`, diga em que grupo e linha este é procurado na cache e como é verificado se lá está ou não (ou seja, se é um *hit* ou um *miss*).

Desl no bloco = 01011 -> 11
Num grupo = 0001000 -> 8
Chave = 11010110
Vai procurar no grupo 8 a chave 11010110. Se encontrar alguma linha com essa chave: hit
Senão: miss

f) Qual é a desvantagem das memórias cache de mapa directo? De que forma essa desvantagem é colmatada pelas memórias cache associativas por grupos?

Se mapa directo, cada bloco tem a posição pré-definida e fixa na cache. Ora pode acontecer que os acessos à memória usem blocos que correspondem às mesmas linhas na cache provocando trocas de blocos e caches misses, quando podem existir muitas outras linhas livres na cache.

6- 1 val

a) Uma arquitectura com suporte para memória virtual baseada em paginação, as tabelas de páginas de todos os processos estão ... (escolha penas uma):

1. Na MMU que está na memória central
2. Na MMU e não na memória central

-> **3. Na memória central**

4. No TLB (*translation look-aside buffer*)

b) Suportando paginação por pedido, ao resolver um endereço virtual no real, pode acontecer que ... (escolha apenas uma):

1. Ocorra uma interrupção por falta de página e o *hardware* (CPU e controlador de disco) vai automaticamente ao disco buscar a página em falta

-> **2. Ocorra uma interrupção por falta de página e o Sistema de Operação tenha de verificar se o programa pode ou não continuar**

3. A respectiva entrada na tabela de páginas não indique uma *frame* (página real), sinal de que o programa acedeu a memória que não é sua e tenha de ser terminado pelo Sistema de Operação
4. A respectiva entrada na tabela de páginas não indique uma *frame* (página real) mas esta possa ser encontrada e resolvida usando o TLB (*translation look-aside buffer*)

7- 4,5 val

Uma arquitectura com endereços virtuais de 28 bits, suporta páginas com dimensão 4 KBytes (2^{12}) e permite paginação a pedido.

a) Diga como é interpretado um endereço virtual para obter a página. Indique que bits representam a número de página.

Num página = Endereço / 4K

12 bits menos significativos: deslocamento dentro da mesma página;

16 bits mais significativos: é o número de página

b) Para um dado processo em execução, o conteúdo da TLB e as primeiras entradas da tabela de páginas são as seguintes (endereços em hexadecimal):

TLB:

página virtual	página física
3	0x000
1	0xAFE
2	0xADA
6	0xEAD

página virtual	página física
0	*inválida*
1	0xAFE
2	0xADA
3	0x000
4	*inválida*
5	0x800
6	0xEAD

As outras entradas são todas inválidas. Indique, justificando, para cada um dos acessos aos endereços virtuais seguintes, se resultam num "*page fault*" ou não e, quando possível, qual o endereço real obtido.

0x0002000 → página 2, está na TLB, end real = 0xADA000

0x0006100 → página 6, está na TLB, end real = 0EAD100

0x0005010 → página 5, está definida na tabela em memória, end real = 0x800010

0x0000020 → página 0, é inválida (não tem frame atribuída), logo, page-fault!

c) Diga, justificando, se os acessos a memória da alínea anterior, que não resultam em "*page fault*", demoram aproximadamente o mesmo tempo ou não.

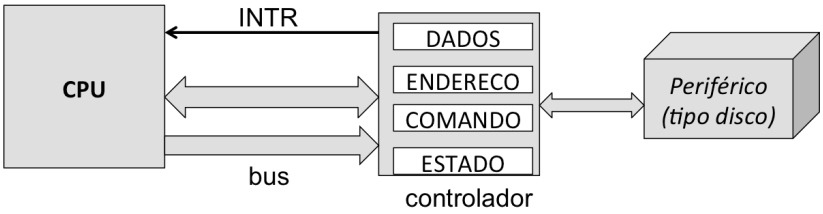
Não porque no acesso a 0x0005010 há necessidade de mais um acesso à memória para obter a página física (frame). Este deve demorar aproximadamente o dobro dos restantes acessos resolvidos na TLB.

8- 5 val

As várias alíneas desta pergunta supõem um ambiente, do ponto de vista do software, semelhante ao usado nas aulas práticas e no mini-projecto, incluindo o TurboC com as seguintes funções:

```
unsigned char inportb (unsigned int portid)
void outportb(unsigned int portid, unsigned char value)
void enable(void)
void disable(void)
setvect( int intrnum, void interrupt(*isr)() )
```

Do ponto de vista do *hardware*, considere o controlador do periférico XYZ, semelhante a um disco, organizado em blocos de 512 bytes cada. Este é o único controlador ligado à linha de interrupção do CPU. Este usa interrupções vectorizadas, sendo para este periférico usado o número 5 para identificar a interrupção.



Considere apenas leituras ao periférico. Todos os registos do controlador têm 8 bits e o bit 7 é o mais significativo. Estes são os seguintes:

- **endereço 0x30 DADOS**: registo que só pode ser lido. O valor lido deste registo é o próximo byte do último bloco lido do periférico. Leituras sucessivas deste endereço devolvem o próximo byte ainda não lido; tal deve ser repetido para obter todos os bytes do bloco.
- **endereço 0x31 ENDERECO**: registo só de escrita; Número do bloco a ser lido.
- **endereço 0x32 COMANDO**: registo só de escrita; Quando o CPU escreve neste registo o valor 0x01, o controlador desencadeia a leitura do bloco com de número indicado em ENDERECO, sem usar interrupções. Quando é escrito o valor 0x02 é também desencadeada a leitura de um bloco mas, quando termina, envia uma interrupção ao CPU. Note que se o controlador já estiver ocupado a ler um bloco estes comandos são ignorados.
- **endereço 0x33 ESTADO**: registo só de leitura. Depois de ser pedida a leitura de um bloco, o controlador colocará o bit 0 a 1 indicando que está ocupado; quando termina a leitura este passa a zero e o bit 1 fica a 1 até que todos os bytes desse bloco sejam lidos via registo DADOS.

a) Escreva, usando o Turbo C, uma rotina com o protótipo

```
void  LeBloco( unsigned int endereco, unsigned char block[512] )
```

Esta pede a leitura do bloco de endereço indicado e aguarda, usando espera activa, a sua leitura devolvendo depois em *block* todos os bytes do bloco pedido.

```
void  LeBloco( unsigned int endereco, unsigned char block[512] )
{
        // se só esta função aceder ao periférico não é necessário esperar para fazer pedido

        outportb( 0x31, endereco );
        outportb( 0x32, 0x01 );
        while ( inportb( 0x33 ) & 1 == 1 )
                ;
        i = 0;
        while ( inportb( 0x33 ) & 2 == 2 )    // tb podem ler 512 bytes
                block[ i++ ] = inportb( 0x30 );

}
```

b) Escreva, usando o Turbo C, uma rotina com o protótipo

```
void  LeBlocoI( unsigned int endereco )
```

Esta pede a leitura do bloco de endereço indicado mas não aguarda pela sua leitura. O fim da leitura deverá ser assinalada pelo controlador com uma interrupção. Note que um programa pode usar esta função ao longo da sua execução para obter vários blocos.

```
void  LeBlocoI( unsigned int endereco )
{
        while ( (inportb( 0x33 ) & 1) == 1 || (inportb( 0x33 ) & 2) == 2 )
                ;

        outportb( 0x31, endereco );
        outportb( 0x32, 0x02 );
}
```

c) Escreva, usando o Turbo C, uma rotina para atendimento de interrupções deste controlador

```
void interrupt meuISR( )
```

Nesta rotina, cada novo bloco lido deve ser deixado num *buffer* global usando a função:

```
void addBuffer( unsigned char blk[512] )
```

```
void interrupt meuISR( )
{
        unsigned char block[ 512 ];
        int i = 0;
        while ( inportb( 0x33 ) & 2 == 2 )    // tb podem ler 512 bytes
                block[ i++ ] = inportb( 0x30 );

        addBuffer( block );
}
```

d) Apresente agora o código que deve ser executado antes da ocorrência da 1ªinterrupção proveniente do periférico, relativamente a inicializações necessárias para que seja possível usar a solução da alínea b) e c). Justifique.

```
Setvect( 5, meuISR );
```

Para colocar no vetor de interrupções o endereço da rotina (meuISR) que atende esta interrupção (intr. núm. 5)