UNIVERSIDADE DA BEIRA INTERIOR
Departamento de Informática

SISTEMAS OPERATIVOS

Exame 1, 4 de Julho de 2018, 14.30 Horas Escala 0:20 Sem Consulta Sala 6.04/6.06 Duração: 2h30m

Observações: Não é permitida a consulta de livros ou de apontamentos. Não se esclarecem dúvidas durante a prova. Se tiver dúvidas, indique na folha de teste a sua interpretação. Utilize uma caligrafia legível.

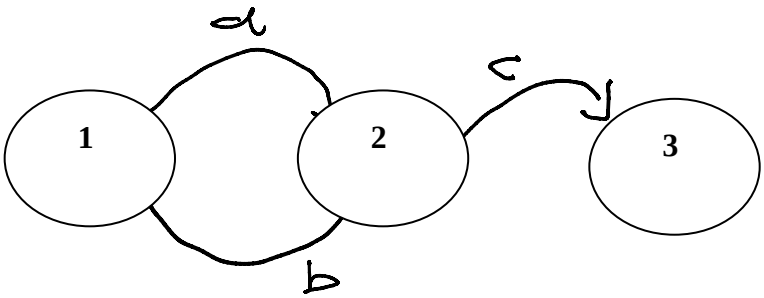
Grupo A: História, Estrutura e Arquitectura dum Sistema Operativo (4 valores)

1. Explique os termos "multiprogramação" e "multiprocessamento" no contexto dum sistema operativo.
2. Para qual tipo de operações é o mecanismo conhecido como "DMA" útil? Exemplifique e explique.
3. Para que servem as *systems calls* (Chamadas ao Sistema)? Qual é o *Modo* em que são executadas?
4. As funções seguintes utilizam ou não chamadas ao sistema : read(), getchar() e sqrt() definidas nos ficheiros <unistd.h>, <stdio.h> e <math.h>respetivamente ? Explique.

Grupo B: Programas, Processos e Threads(4 valores)

5. Considere o seguinte diagrama muito básico dos estados dos processos num sistema operativo

   - Escreva uma legenda para o diagrama - não se esqueça de indicar o nome dos estados {1,2,3} bem como explicar as transições {a,b,c}.
   - No diagrama não existe uma transição entre o estado 3 e o estado 1 nem o estado 3. Porqué é que é necessário um estado como este ?
   - O diagrama não inclui o estado para os processos descritos em Linux como em estado *"Bloqueados"*. Qual o objetivo deste estado? Faça um novo diagrama para incluir este estado



6. Qual a relação entre "Processos", "Threads" e "Jobs" no sistema operativo Linux? Explique.
7. Considere um Sistema Operativo cujo algoritmo de escalonamento é FCFS (first com, first served) sem preempção, qual é o output do seguinte programa ? (Deverá mostrar o *trace* do funcionamento do programa) .

```
int main()
{
   int  pid, x = 5;
   pid = fork();
   if (pid == 0) { fork(); x=x-2; }
   else          { x++; }
   printf("x=%d\n",x);
}
```

Grupo C: Gestão de Memória (4 Valores)

1. *Data Execution Prevention* (DEP) é uma funcionalidade de segurança incluída em alguns sistemas operativos cujo objetivo é o de impedir a execução de instruções vindas de certas regiões da memória duma aplicação. Explique como é que este mecanismo poderia ser implementado usando um sistema de memória paginada e dê exemplos das áreas de memoria dum processo a proteger.

2. Um sistema de memória virtual tem um tamanho de página de 32 palavras, 8 páginas virtuais e 3 páginas físicas. Um endereço virtual neste sistema tem 8 bits, sendo que os primeiros 3 bits indicam o número de página. A tabela de páginas está inicialmente no estado apresentado em baixo:

Página virtual	Página física	*Valido/Invalido*
0	1	*1*
1	2	*1*
2		*0*
3		*0*
4		*0*
5		*0*
6		*0*
7	0	1

i. 111 00010
ii. 000 00011
iii. 001 01100
iv. 010 10000
v. 000 01111
vi. 011 01000
vii. 100 11100

a) Indique se o resultado dos endereços lógicos indicados em cima e referenciados por esta ordem decrescente é uma page hit (sucesso), uma page miss (falha) ou uma trap (uma interrupção devido um erro).
Se for uma page-miss (falha) será **invocado** o algoritmo de substituição de paginas (LRU-Least Recently Used / Menor usada recentemente) – quer dizer que a tabela de paginas poderá **mudar**!

b) Calcule o endereço físico resultante (exceto no caso dum trap) em valor decimal.

c) Mostre a tabela de páginas no fim desta sequência de endereços.

Grupo C: Concorrência, Sincronização e Bloqueio (4 valores)

8. Considere um sistema que gere contas bancárias e que serve clientes, identificados através dum número inteiro único, que efetuam pedidos de transferência de dinheiro entre contas do mesmo sistema. Tais pedidos podem ocorrer concorrentemente, sendo cada pedido servido por uma ***thread*** diferente. Assuma os seguintes requisitos fundamentais deste sistema:

   Requisito A: Transferir uma dada quantia duma conta origem, A, para uma conta destino, B, consiste em debitar essa quantia de A, e creditar B com a mesma quantia, e mais nada.

   Requisito B: Uma dada transferência entre as conta A e B deve parecer **atómica** do ponto de vista de outras transferências que observem o saldo de A ou de B.

   Requisito C: Se, concorrentemente, um cliente pedir para transferir de A para B, e outro cliente pedir para transferir de C para D (em que A, B, C e D são contas diferentes), as duas transferências devem correr completamente em paralelo (i.e. sem que nenhuma seja obrigada a esperar pela outra).

   Requisito D: O sistema não deve chegar a situações de deadlock.

Considere as seguintes implementações da função ***transfere*** (que cada thread executa para servir cada pedido de transferência):

```
transfere1(int A, int B, int quantia) {
   saldo[A] -= quantia;
   saldo[B] += quantia;
}
```

```
pthread_mutex_t trinco;

transfere2(int A, int B, int quantia) {
   pthread_mutex_lock( &trinco )
    saldo[A] -= quantia;
    saldo[B] += quantia;
   pthread_mutex_lock( &trinco )
}
```

```
pthread_mutex_t trincos[MAX_NUMERO_CONTAS];

transfere3(int A, int B, int quantia) {
   pthread_mutex_lock( &trincos[A] )
   pthread_mutex_lock( &trincos[B] )
     saldo[A] -= quantia;
     saldo[B] += quantia;
   pthread_mutex_unlock( &trincos[A] )
   pthread_mutex_unlock( &trincos[B] )
}
```

Assuma que a instrução alto-nível "uMsaldo -= quantia" ("uMsaldo += quantia")
corresponde à seguinte sequência de instruções máquina em escritas em assembler NASM:
mov AX, uMsaldo
mov BX, quantia
sub AX, BX (ou add AX, BX)
mov uMsaldo, AX

- Para cada implementação, qual/quais dos requisitos não são satisfeitos? Ilustre com um exemplo cada requisito que afirmar não ser satisfeito. No caso de haver DeadLock deve ser feito um diagrama de alocação de recursos.

- Altere a solução ***transfere3*** para que, no caso da conta de origem não ter saldo suficiente, a transferência se **bloqueie** até que tal aconteça. Mas, não deve utilizar nem semáforos nem mutexes.

  Comente a sua solução em termos de desempenho e funcionamento global do sistema.

Grupo E: Exercício Prático (4 valores)

9. Utilizando pipetas de baixo nível é possível criar um canal de comunicação entre dois processos. Por exemplo é possível implementar a funcionalidade do "pipe" do bash Shell fechando os descritores de ficheiro correto e associando-as novamente a um dos lados duma pipeta.

   Neste exercício escreverá um programa para executar concorrentemente dois programas, utilitários do Shell, nomeadamente os programa **curl** e **grep**. O programa **curl** ligar-se-á a um *endereço* da Internet e vai buscar notícias sobre o mundial 2018 e o programa **grep** filtrará estas noticias e imprimirá as linhas que contêm referência a por exemplo a equipa *Brazil*. O endereço da Internet e nome da equipa para pesquisar e filtrar são passados como parâmetros, o primeiro parâmetro passado ao programa conterá o endereço da Internet e o segundo o nome da equipa.

   Exemplo: >./meuPrograma http://ww.fifa.com/worldcup/news/ Portugal

```
/* meuPrograma.c
    Objectivo: Executar concorrentemente  o comando: curl -c endereço | grep equipa
*/
main(int argc, char *argc[])  {
   if (argc!=3) return 1;
   char *p1[4]={"curl","-c",NULL,NULL};
   char *p2[3]={"grep",NULL,NULL};

   ……

   return 0;
}
```

Na sua folha de teste complete o programa. Vai precisar das funções close(), pipe(),dup(), fork(), execvp() cujas sintaxes são dadas numa folha anexa.

Não é necessário verificar o sucesso ou não das chamadas ao sistema.

Nota que :
curl -c http://ww.fifa.com/worldcup/news/ : imprimir output do endereço no stdout
grep Brazil : imprimir (no stdout) as linhas do standard input que contêm a palavra Brazil