



# FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

## Mestrado em Engenharia Informática e Computação

SISTEMAS OPERATIVOS (EIC0027) – 2012/2013 - 2º semestre

Exame da Época Normal

13/Junho/2013

### PARTE A – sem consulta

Duração: 15 minutos

NOME DO ESTUDANTE: \_\_\_\_\_ Nº: *EI* \_\_\_\_\_

1. [4.0]

Indique quais das seguintes afirmações são verdadeiras e quais são falsas, assinalando respectivamente com **V** ou **F**:

	Um programa a ser executado num computador em que o escalonamento é preemptivo terá um tempo total de execução menor do que se o escalonamento fosse não-preemptivo.
	Durante uma operação de comutação de contexto entre dois processos é necessário guardar o conteúdo do <i>program counter</i> e dos restantes registos do processador.
	Se um mesmo ficheiro for aberto por dois processos, P1 e P2, o descritor do ficheiro retornado a P1 será igual ao descritor retornado a P2.
	A melhor forma que um programador tem de prevenir a existência de <i>deadlocks</i> nos seus programas é impedir a ocorrência de "espera circular".
	O "princípio da localidade de referência" é o fundamento da técnica de gestão de memória virtual e também da chamada "estratégia dos conjuntos de trabalho" ( <i>working set strategy</i> ) usada para evitar o <i>thrashing</i> .
	Após uma chamada <b>fork()</b> o processo-filho recebe uma cópia das variáveis globais, mas não das variáveis locais do processo-pai.
	A ocorrência de processos <i>zombie</i> deve ser evitada pois estes processos podem interferir negativamente com outros processos em execução.
	Em Linux, um <i>FIFO (named pipe)</i> pode ser usado para pôr em comunicação dois processos a executarem em computadores diferentes, desde que ambos conheçam o nome do <i>FIFO</i> .
	Uma das limitações do uso de sinais como mecanismo de comunicação entre processos é que não há garantia que todos os sinais enviados a um processo sejam recebidos por este.
	Tendo em conta o protótipo de <code>pthread_create()</code> , <code>int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void * (*func)(void *), void *arg)</code> a seguinte instrução <code>for(t=0; t&lt;N; t++) pthread_create(&amp;tid[t], NULL, threadFunc, &amp;t);</code> criará <i>N threads</i> , cada uma executando a função <code>threadFunc()</code> ; cada <i>thread</i> receberá, como parâmetro, um valor diferente de <i>t</i> .



FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

**Mestrado em Engenharia Informática e Computação**

SISTEMAS OPERATIVOS (EIC0027) – 2012/2013 - 2º semestre

Exame da Época Normal

13/Junho/2013

**PARTE B – com consulta**

**Duração: 2 horas**

**NOME DA(O) ESTUDANTE:** \_\_\_\_\_ **Nº:** *EI* \_\_\_\_\_

**2.**

**a)** [0.7] Explique brevemente a importância do mecanismo de *DMA (Direct Memory Access)* para a implementação da multiprogramação.

**b)** [0.7] Em muitos sistemas operativos a prioridade dos processos dos utilizadores varia dinamicamente durante a sua execução. Apresente razões para esta variação.

**c)** [0.9] Indique 3 formas diferentes de 2 processos independentes (sem qualquer relação de "parentesco") comunicarem entre si, em Unix/Linux. Compare-as brevemente quanto ao tipo e volume de informação que pode ser transferida.


--

Considere o seguinte extrato de um programa:

```

1  __ redi rectStdin ( _____ ) {
2      ...
3  }
4
5  int main() {
6      int numValuesRead, value;
7
8      if (redi rectStdin ("infile.txt") < 0 ) {
9          perror ("redi rectStdin");
10         exit (1);
11     }
12
13     numValuesRead = scanf ("%d", &value);
14     if (numValuesRead == 1) {
15         printf ( "'value' = %d was read from 'infile.txt'\n", value);
16         exit(0);
17     }
18     else {
19         printf ( "Error reading 'value' from 'infile.txt'\n");
20         exit(1);
21     }
22 }

```



**b) [0.5]** Indique as alterações a fazer na função **main()** de modo a que a entrada standard possa ser redirecionada para o ficheiro que for passado ao programa como primeiro argumento da linha de comando. NOTA: basta indicar as linhas a alterar, usando a numeração indicada na margem esquerda do código fornecido.

**4.**

Considere o seguinte código que implementa, em linguagem *C-like*, uma solução para o conhecido problema de sincronização dos "leitores e escritores, com prioridade aos leitores", analisado nas aulas teóricas. Considere uma situação em que:

- chegaram leitores (**Ri**) e escritores (**Wi**) na seguinte sequência (**t** indica o tempo, em unidades **ut**): **R1**, em **t=0**; **W1**, em **t=1**; **R2**, em **t=2**; **W2**, em **t=7**; **R3**, em **t=8**;
- o tempo gasto numa operação de leitura, **readUnit()**, é de **4 ut**;
- o tempo gasto numa operação de escrita, **writeUnit()**, é de **5 ut**;
- o tempo gasto nas restantes operações é desprezável, face aos tempos de leitura/escrita da informação.

```
sem_init(&wSem, 1);
sem_init(&X, 1);
```

```
// -----
```

```
void writer()
{
    sem_wait(&wSem);
    writeUnit();
    sem_post(&wSem);
}
```

```
void reader()
{
    static int readCount = 0;
    sem_wait(&X);
    readCount++;
    if (readCount==1) sem_wait(&wSem);
    sem_post(&X);
    readUnit();
    sem_wait(&X);
    readCount--;
    if (readCount==0) sem_post(&wSem);
    sem_post(&X);
}
```

**a) [1.0]** Desenhe, no quadro abaixo, com recurso aos símbolos indicados, a evolução do estado dos leitores e dos escritores, assinalando os períodos de espera de cada leitor/escritor e os períodos em que acederam à informação.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
R1	>																									
R2			>																							
R3									>																	
W1		>																								
W2								>																		
>	- chegada / arrival																									
.....	- espera / waiting																									
<div></div>	- acesso à informação / information access																									

**b) [0.5]** Diga qual o estado de cada um dos semáforos no instante **t=10 ut**, indicando que processo tem a posse do semáforo e que processo(s) está(ão) bloqueado(s) na respetiva fila.

NOME DA(O) ESTUDANTE: \_\_\_\_\_ Nº: 21

**Nota:** no código solicitado nas perguntas 5 e 6, pode omitir as diretivas de inclusão e os testes de erro nas chamadas ao sistema, exceto nas situações indicadas

5.

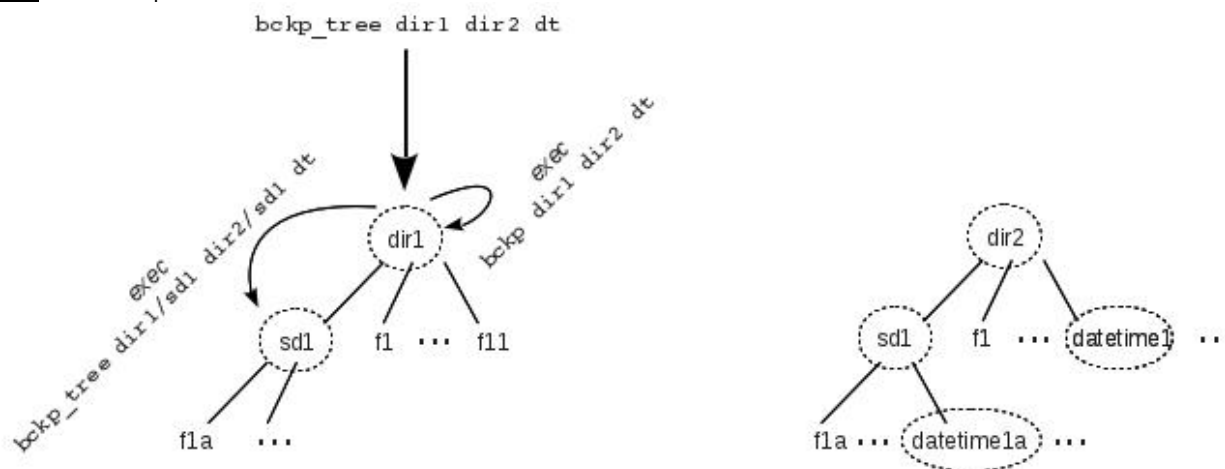
Relembre o 1º projeto desta unidade curricular, "backup & restore of files", para um sistema Unix e suponha que, no seu âmbito, foi desenvolvido um programa **bckp** perfeitamente funcional que efetua o previsto: invocado como

mantém em **di r2** uma cópia de segurança dos ficheiros regulares de **di r1**, atualizada em intervalos de tempo **dt**.

Considere agora que pretende aproveitar esse programa para efetuar a cópia de salvaguarda dos ficheiros de todo um diretório, independentemente do seu tamanho e complexidade (ver figura). Para o conseguir, vai desenvolver um programa, **bckp\_tree**, que vai percorrer um dado diretório, nele invocar **bckp** e, em cada um dos seus sub-diretórios, lançar uma cópia de si, **bckp\_tree**, para dar continuidade à salvaguarda. O novo programa **bckp\_tree**, seria invocado, de forma semelhante a **bckp**, como

e dentro de **dl r2** ficarão também os sub-diretórios respeitantes ao "backup" de cada sub-diretório de **dl r1**, construídos de forma idêntica ao definido no 1º projeto da unidade curricular.

**NOTA:** assumo que **di r2** nunca é um sub-diretório de **di r1**.



Um possível esquema da operação de **bckp\_tree** tem as seguintes fases:

1. recolher os parâmetros da linha de comando;
2. invocar **bckp** em **dl r1**;
3. percorrer os ficheiros de **dl r1** e, para cada sub-diretório encontrado, **sdi**, invocar nova instância de **bckp\_tree** com esse diretório como 1º parâmetro, **dl r1/sdi**, e como 2º parâmetro, **dl r2/sdi**.

a) [1.3] Escreva o código da fase 2 do esquema de **bckp\_tree**, tomando razoáveis precauções para detetar situações de erro.

--

**b) [1.3]** Defina as sub-fases da fase 3, em português ou em pseudo-código.

[illegible]



NOME DA(O) ESTUDANTE: \_\_\_\_\_

Nº: *EI*

6.

Pretende-se desenvolver um programa *multithreaded* que copie os dados (sequências de caracteres) que lhe chegam através de **n** FIFOs de entrada para um único FIFO de saída (um *multiplexador*). O programa deverá ser invocado da seguinte forma:

**mux f1 f2 ... fn fout**

onde **f1, f2, ..., fn** são os nomes dos FIFOs de entrada e **fout** é o nome do FIFO de saída. Assume-se que os FIFOs já estão criados no sistema.

Cada FIFO de entrada tem associada uma *thread*, **receive**, que lê, um a um, os caracteres do FIFO e os coloca num *buffer* circular, comum. Os caracteres são retirados deste *buffer* por uma outra *thread*, **send**, que os envia para o FIFO de saída, pela mesma ordem em que foram colocados no *buffer*.

**Exemplo:** se uma aplicação enviar a sequência de caracteres "123" para o FIFO **f1** e outra aplicação enviar a sequência "abcd" para o FIFO **f2**, o que é enviado para o FIFO **fout**, pode ser "123abcd", ou "a12b3cd", ou "abc1d23", ou ..., isto é, qualquer combinação de caracteres que mantenha a ordem relativa inicial dos mesmos, sem repetição de caracteres.

Análise o código seguinte que implementa parcialmente o referido programa:

```

1  #define N 100    // buffer size
2  char buffer[N];
3  int ri = 0;     // read index
4  int wi = 0;     // write index
5  pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
6  sem_t canl read, canl write;
7
8  int open_in_fifo(char* fifoName) {
9      /* TODO B: opens the FIFO and exits the thread if the open fails,
10         otherwise returns the FIFO descriptor */
11  }
12
13  void* receive(void* p) {
14      int f = open_in_fifo(p);
15      char c;
16      while(read(f, &c, 1)){
17          printf("Read from FIFO %s the value %d\n", (char*)p, c);
18          pthread_mutex_lock(&mut);
19          buffer[wi++] = c;
20          wi %= N;
21          pthread_mutex_unlock(&mut);
22      }
23      close(f);
24  }
25
26  int open_out_fifo(char* fifoName) {
27      /* Opens the FIFO and exits the program if it fails,
28         otherwise returns the FIFO descriptor */
29  }
30
31  void* send(void* p) {
32      int f = open_out_fifo(p);
33      for(;;){
34          printf("Write to FIFO %s the value %d\n", (char*)p, buffer[ri]);
35          write(f, &buffer[ri++], 1);
36          ri %= N;
37      }
38      close(f);
39  }
40
41  void init_semaphores() {
42      //TODO C: Initialize both semaphores with the appropriate values
43  }
44
45  void destroy_semaphores() {
46      sem_destroy(&canl read);
47      sem_destroy(&canl write);
48  }
49
50  void main(int argc, char** argv) {
51      pthread_t st;    // send thread
52      pthread_t* rt;   // receive threads
53      if (argc < 3){
54          perror("Usage ./mux f1 [f2 [...[fn]]] fout\n");
55          exit(1);
56      }
57      init_semaphores();
58      /* TODO A: creates receive() threads, one for each input FIFO and
59         one send() thread, as well any other required code */
60      free(rt);
61      destroy_semaphores();
62  }

```

[illegible]

--

--

[illegible]