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 *program counter* 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 *deadlocks* 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" (*working set strategy*) usada para evitar o *thrashing*.
	Após uma chamada **`fork()`** o processo-filho recebe um cópia das variáveis globais, mas não das variáveis locais do processo-pai.
	A ocorrência de processos *zombie* deve ser evitada pois estes processos podem interferir negativamente com outros processos em execução.
	Em Linux, um *FIFO* (*named pipe*) pode ser usado para pôr em comunicação dois processos a executarem em computadores diferentes, desde que ambos conheçam o nome do *FIFO*.
	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 pthread_create(), **`int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void * (*func)(void *), void *arg)`** a seguinte instrução **`for(t=0; t<N; t++) pthread_create(&tid[t], NULL, threadFunc, &t);`** criará **`N`** *threads*, cada uma executando a função **`threadFunc()`**; cada *thread* receberá, como parâmetro, um valor diferente de **`t`**.

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:** $\mathcal{E}\mathcal{I}$_____

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.

d) [0.7] A forma mais utilizada para a gestão de memória num sistema operativo moderno é a paginação a pedido (*demand paging*). Dois fatores que podem afetar a performance da paginação a pedido são a "*hit ratio*" e a "taxa de falta de páginas" (*page fault rate*). Explique sucintamente o significado destes termos e como é que eles influenciam a performance da paginação a pedido.

3.
Considere o seguinte extrato de um programa:

```
1   ___ redirectStdin (____________________) {
2      ...
3   }
4
5   int main() {
6     int numValuesRead, value;
7
8     if (redirectStdin ("infile.txt") < 0 ) {
9       perror ("redirectStdin");
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  }
```

a) [1.0] A função **redirectStdin()** deve redirecionar a entrada standard do processo para o ficheiro que lhe for passado como parâmetro, retornando **-1** se ocorrer algum erro ou **0** (zero) se tiver sucesso em todas as chamadas que efetuar ao sistema. Analise a função **main()** para perceber melhor o uso de **redirectStdin()**. Escreva o código desta função.

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*

▒ - 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:** $\mathcal{EI}$_____

***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

```
bckp dir1 dir2 dt
```

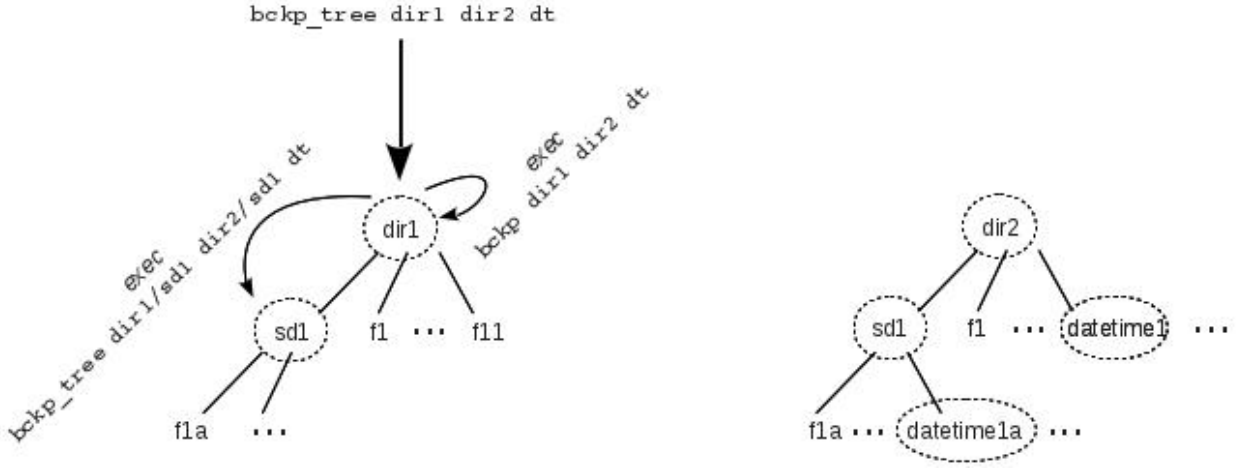
mantém em **dir2** uma cópia de segurança dos ficheiros regulares de **dir1**, atualizada em intervalos de tempo **dt**.

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

```
bckp_tree dir1 dir2 dt
```

e dentro de **dir2** ficarão também os sub-diretórios respeitantes ao "backup" de cada sub-diretório de **dir1**, construídos de forma idêntica ao definido no 1º projeto da unidade curricular.

NOTA: assuma que **dir2** nunca é um sub-diretório de **dir1**.



Um possível esquema da operação de **bckp_tree** tem as seguintes fases:

1. recolher os parâmetros da linha de comando;
2. invocar **bckp** em **dir1**;
3. percorrer os ficheiros de **dir1** e, para cada sub-diretório encontrado, **sd**i, invocar nova instância de **bckp_tree** com esse diretório como 1º parâmetro, **dir1/sd**i, e como 2º parâmetro, **dir2/sd**i.

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.

c) [2.4] Escreva o código da fase 3, supondo que as chamadas ao sistema são bem sucedidas.

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 carateres do FIFO e os coloca num *buffer* circular, comum. Os carateres 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 carateres "**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 carateres que mantenha a ordem relativa inicial dos mesmos, sem repetição de carateres.

Analise 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 canIread, canIwrite;
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(&canIread);
47    sem_destroy(&canIwrite);
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  }
```

a) [1.5] Escreva o código em falta na função **`main()`**, ***TODO A***, de modo a instanciar corretamente as *threads*. Note que cada *thread* **`receive`** só lê de um único FIFO.
Faça também a reserva de recursos que achar necessária e escreva qualquer outro código que seja necessário para o correto funcionamento do programa. Ignore, por enquanto, as questões de sincronização.

b) [1.0] Escreva o código da função **`open_in_fifo()`**, ***TODO B***, para abrir, em modo de leitura, um FIFO cujo nome é recebido como parâmetro. Em caso de sucesso deve devolver o descritor do FIFO; em caso de erro deve terminar a *thread*.

c) [1.0] Justifique a necessidade das chamadas das linhas **18** e **21** do código apresentado anteriormente.

d) [1.5] Tendo em vista a correta sincronização no acesso ao *buffer*, e evitar situações de *overflow* ou *underflow*, escreva o código da função **`init_semaphores()`**, ***TODO C***, para inicializar adequadamente os semáforos. Escreva também o código em falta, algures nas funções **`receive()`** e **`send()`**, necessário para sincronizar o acesso ao *buffer*.
NOTA: basta escrever o código adicional destas duas funções, indicando com recurso à numeração das linhas o respetivo local

FIM