MIEEC **Sistemas Operativos – 2015/16** **FEUP**

Tipo de Prova: consulta de "*cheat-sheet*" Possível solução do **Exame da Época Normal**
Duração: 2 horas **22.Junho.2016**
Cotação máxima: 20 valores
Estrutura da prova: Parte I (escolha múltipla, 30%); Parte II (mais convencional, 50%); Parte III (predominantemente de aplicação, 20%)

PARTE I: Escolha múltipla [6 val.]

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	B	A	B	A	C	B	B	C	B	A	C	C	A

PARTE II: Mais convencional [10 val.]

1. [1 val.]

1º: `host-sh$ usocat pipe - > file`
2º: `guest-sh$ cat device.c > /dev/serial`
3º: `host-sh$ diff file shared/device/device.c`

2. [1 val.]

No código `seri.c` :
em `seri_init()`:

```
   ...
   int err = request_irq(SERIAL_INT, function, FLAGS, MOD_NAME, dev_id);
        // SERIAL_INT – numinterrupção porta série
        // function: função a invocar pelo sistema aquando de uma interrupção
        // FLAGS: usar em certas situações, e.g. interrupção partilhada. Aqui,
põe-se a 0!
        // MOD_NAME: nome do módulo, e.g "seri"
        // dev_id:  identificação do dispositivo (minor)
   ...
   outb(UART_IER_RDI | UART_IER_THRI, SERIAL_BASEPORT+UART_IER);
        // UART_IER_RDI: bit ou bits da interrupção de recepção
        // UART_IER_THRI: bit ou bits da interrupção de transmissão
        // SERIAL_BASEPORT+UART_IER: porta de activação de interrupções
   ...
```

em `seri_exit()`:

```
   ...
   outb(0x00, SERIAL_BASEPORT+UART_IER);
        // 0x00: bits de desactivação de interrupções
   free_irq(SERIAL_INT, dev_id);
        // args iguais aos de request_irq()
   ...
```

3. [1 val.]

```
int flag = 0; // init
...
while (TestAndSet (&flag, 1) != 0)
 schedule(); //busy wait
/* …
Zona Crítica
… */
flag = 0;
```

4. [1 val.]
Suponhamos que `number` tem o valor `MAX_VAL` e que após Thread 1 executar `unlock()` termina o seu quantum de CPU e é retirado de execução; se o Thread 2 tiver tempo de executar as instruções até 6, antes de Thread 1 retomar a execução, o `wakeup()` é perdido, pois `sleep()` não foi ainda executado; qdo o Thread 1 fizer `sleep()` poderá adormecer para sempre.

5. [1 val.]
Escolher duas entre:
- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

6. [1 val.]

a) Não, pois no código não se obriga o processo-filho a executar antes do pai.

b)

```
1.  //#include . . .
2.
3.  void *function (void *argum) {
4.     printf("Hello, ");
5.     return (NULL);
6.  }
7.
8.  int main() {
9.        pthread_t tid;
10.    pthread_create(&tid, NULL, function, NULL);
11.    pthread_join(tid, NULL);
12.    printf("world!");
13.    return (0);
14.  }
```

7. [1 val.]

a) É a pág. 97 (actual/ no quadro 3), pois é a que foi referenciada (usada) há mais tempo, há 250 clock ticks.

b) Vantagem: é dos algoritmos que tem taxas de sucesso maiores, pois em geral uma página que não é usada há muito tempo, não o será tão cedo.
Desvantagem: exige maiores recursos contabilísticos, através de um contador de tempo com vários bits que tem de ser actualizado a cada referência de página.

8. [1 val.] Do cap. 18 do livro OSTEP: «*For every memory reference (whether an instruction fetch or an explicit load or store), paging requires us to perform one extra memory reference in order to first fetch the translation from the page table*.»

9. [1 val.]

a) Apesar da rapidez dos discos actuais, grande velocidade de rotação e movimento dos braços de leitura em conjunto, podendo ler de um cilindro (grupo de pistas), o facto é que a componente mecânica ainda é determinante na limitação dos tempos de acesso, comparada com a componente electrónica. Assim sendo, t_{seek}, $t_{rotation}$, são os factores preponderantes, comparados com $t_{transfer}$ na rapidez de acesso.

b) É mínima, pois não tem conhecimento do posicionamento físico dos blocos de dados no disco. Eventualmente, poderá agrupar os pedidos de blocos de forma a _tentar_ facilitar (segundo uma ideia que só pode ser aproximada) a recolha da informação do disco real ou a escrita nele.

10. [1 val.]

a) nº de blocos acessíveis * tamanho do bloco =
= (8 endereços_directos + 1 endereço_indirecto * (4KiB / 4B)) * 4KiB = (8 + 1024) * 4KiB = 4128 KiB
Nota: (4KiB / 4B) é o nº de endereços de bloco que cabe num bloco

b) Só é necessário guardar em memória os i-nodes dos ficheiros abertos, ao contrário da FAT, que tem de ter em memória entradas referentes a todos os blocos do disco.

PARTE III: Predominantemente de aplicação [4 val.]

1. [2 val.]

a) `maiusc_efic()`:

```
1.  #define MAX 1024
2.  void maiusc_efic(int fd1, int fd2) {
3.            char buf[MAX];
4.            int n, i;
5.       while ((n = read(fd1, buf, MAX)) > 0) {
6.            for (i=0; i<n; i++)
7.                 buf[i] = (char) toupper(buf[i]);
8.            write(fd2, buf, n);
9.            }
10.      if (n < 0)
11.           perror("read");
12.   }
```

b) `prog2.c`, `main()`:

```
1.  int main(int argc, char *argv[], char *env[]) {
2.     int fd1, fd2;
3.  if (argc != 3) {
4.     printf("\nUso: %s <fich1> <fich2>\n", argv[0]);
5.     exit(0);
6.     }
7.  if ((fd1 = open(argv[1], O_RDONLY)) < 0) {
8.     perror("open fd1");
9.     exit(1);
10.    }
11.  if ((fd2 = open(argv[2], O_WRONLY | O_TRUNC |O_CREAT, 0644)) < 0) {
12.    perror("open fd2");
13.    exit(1);
14.    }
15.  maiusc(fd1, fd2);
16.  return 0;
17.  }
```

c) `prog3.c`, `main()`:

```
1.  int main(int argc, char *argv[], char *env[]) {
2.     int fd1, fd2;
3.
4.  if (argc != 3) {
5.     printf("\nUso: %s <fich1> <fich2>\n", argv[0]);
6.     exit(0);
7.     }
8.  if ((fd1 = open(argv[1], O_RDONLY)) < 0) {
9.     perror("open fd1");
10.    exit(1);
11.    }
12.  if ((fd2 = open(argv[2], O_WRONLY | O_TRUNC |O_CREAT, 0644)) < 0) {
13.    perror("open fd2");
14.    exit(1);
15.    }
16.  dup2 (fd1, STDIN_FILENO);
17.  dup2 (fd2, STDOUT_FILENO);
18.  if (execlp("prog1", "prog1", NULL) < 0) {
19.    perror ("execlp prog1");
```

```
20.     exit(2);
21.     }
22.   return 0;
23.   }
```

2. [2 val.]

a) Da listagem de `calc.c`:

i. Imediatamente antes de li. 26 escrever:

```
free(argum);    // i. do not allow memory leaks!
```

Imediatamente antes de li.35 escrever:

```
free(pdata);    // i. do not allow memory leaks!
```

ii. comentar li. 38:

```
// pthread_join(pdata->id, NULL); ii. join() is invalid for detached threads!
```

iii. Antes das funções, introduzir:

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER; // iii. synchronize shared
vars
```

Substituir li. 24 por:

```
pthread_mutex_lock(&mut); // iii. synchronize shared vars
noper++;
printf("\nTotal de operações (até ao momento): %lu.\n", noper);
pthread_mutex_unlock(&mut); // iii. synchronize shared vars
```

Comentar a li. 40.

NOTA: se se desejar uma versão sem *threads* independentes, era preciso modificar um pouco mais o código para ir guardando os tids e depois, no final, com eles efectuar `join`s. Aliás, no código original, li. 37 e 38, o `join` nunca pode ficar em ciclo logo a seguir ao `create`, sob pena de o programa ficar serializado!

b) Antes das funções, introduzir:

```
#define MAXTHREADS 5
int nthreads = 0;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
```

Substituir li.38-39 por:

```
pthread_mutex_lock(&mut);
nthrs++;
while (nthrs >= MAXTHREADS) {
     pthread_cond_wait(&cv, &mut);
}
pthread_mutex_unlock(&mut);
```

Substituir li. 24 por:

```
pthread_mutex_lock(&mut);
noper++;
nthrs--;
printf("\nTotal de operações (até ao momento): %lu.\n", noper);
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mut);
```

Comentar a li. 40.

JMMC