Teste de Sistemas de Operação — 15 de Abril de 2010

Duração: 1 h (prática + teórica) **D**

NOME: _____

Indique apenas uma das alternativas. Respostas erradas *descontam* na nota.

1. Em um sistema que suporta multiprogramação e *time-sharing*, vários utilizadores podem compartilhar o sistema simultaneamente. Se este suporte não for bem implementado, esta situação pode gerar vários problemas de segurança. Um possível problema de segurança pode ser:

a. processos podem invadir a área de memória de outros processos

b. processos podem executar em modo não protegido

c. processos podem utilizar a CPU de forma caótica

2. Um pedido de entrada e saída sem buffering:

a. demora menos tempo a ser processado

b. demora mais tempo a ser processado

c. não possibilita a sobreposição de I/O com o uso de CPU

3. O mecanismo de interrupções (trap ou software interrupt) é fundamental para

a. executar funções de sistema.

b. executar processos de sistema em modo-batch.

c. implementar uma TLB (Translation Lookaside Buffer).

4. A função de sistema `execl()` permite substituir o programa do processo que executa a função por um outro; isto acontece porque

a. se altera todo o contexto do processo.

b. se modifica o segmento de dados e texto do processo.

c. se cria um processo filho que vai executar o novo programa dado como argumento na função `execl()`.

5. Qual dos seguintes códigos resolve o problema de exclusão mútua sobre uma região crítica, entre 2 processos, de forma correta?

a. 1

b. 2

c. 3

(1)

```
int turn;
turn <- 1;
proc(0) AND proc(1);

proc(int i)
{
    while (TRUE)
    {
       compute;
       while (turn <> i);
       <zona crítica>;
       turn <- i+1 mod 2;
    }
}
```

(2)

```
boolen flag[2];
flagp[0] <- flag[1] <- FALSE;
proc(0) AND proc(1);

proc(int i)
{
    while (TRUE)
    {
        compute;
        while (flag[i+1 mod 2]);
        flag[i] <- TRUE;
        <zona crítica>;
        flag[i] <- FALSE;
     }
}
```

(3)

```
boolean flag[2];
flagp[0] <- flag[1] <- FALSE;
proc(0) AND proc(1);

proc(int i)
{
    while (TRUE)
    {
        compute;
        flag[i] <- TRUE;
        while (flag[i+1 mod 2]);
        <zona crítica>;
        flag[i] <- FALSE;
    }
}
```

6. A função de sistema `kill()` serve:

a. para enviar sinais explícitos a um processo filho.

b. apenas para terminar um processo filho.

c. para terminar o processo que invoca a função.

7. Considere a seguinte frase: "O Unix não é um sistema adequado para aplicações de tempo real porque um processo executando em modo kernel não pode ser interrompido". Esta afirmativa é falsa ou verdadeira?

a. falsa.

b. verdadeira.

c. não há dados suficientes para responder a esta questão.

8. A instrução `fork()` cria um processo filho. Suponha que executou o código abaixo. Podemos dizer que:

a. foram criados 3 processos filho

b. foram criados 5 processos filho

c. foram criados 4 processos filho

```
for (i=0; i<2; i++)
   if (fork()==0) {
      escreve(i);
   }
```

9. Quais destas instruções devem ser executadas em modo privilegiado?

a. Modificar o valor do relógio, ler valor do relógio, desligar interrupções

b. Mudar de modo utilizador para modo kernel, desligar interrupções

c. Limpar a memória, modificar o valor do relógio

10. A função de sistema `wait()` permite uma forma limitada de comunicação entre processos pai e filho, nomeadamente:

a. permite que o pai espere até que um filho em concreto termine.

b. permite que o pai espere até que um filho termine.

c. permite que o filho sincronize com o pai esperando que este termine.

11. A instrução `fork()` cria um processo filho. Suponha que se executou o código abaxo. Podemos dizer que:

a. o processo pai escreve o valor `2` e o processo filho escreve o valor `3`

b. o processo pai escreve o valor `3` e o processo filho escreve o valor `2`

c. apenas o processo pai escreve o valor da variável `x`.

```
if (fork()!=0) x=2; else x=3;
escreve(x);
```

12. Uma instrução que verifique e modifique uma posição de memória de forma atómica

a. é fundamental para implementar sincronização entre processos

b. não é necessária para implementar sincronização entre processos

c. torna a implementação da sincronização entre processos mais fácil

13. O uso de DMA (Direct Memory Access) tem o efeito de?

a. Reduzir o número de vezes que os dados passam no bus do sistema.

b. Reduzir a intervenção do SO na operação de I/O

c. Facilitar o acesso directo dos processos à memória da máquina.

Parte Prática

Considere o excerto de programa que se segue como a implementação da etapa ’matmult’ do enunciado do Trabalho I.

```
 1.  main(int argc, char *argv[]) {
 2.    int i, j, len, L;
 3.    char *line = NULL;
 4.    FILE *stream = fopen(argv[1], "r");
 5.    fscanf(stream, "%d", &L);
 6.    getline(&line, &len, stream);
 7.    int in[2], out[L][2];
 8.    for (i = 0; i < L; i++) {
 9.      pipe(in);
10.      pipe(out[i]);
11.      if (fork() != 0) {
12.        close(out[i][WRITE]);
13.        getline(&line, &len, stream);
14.        // escrita na pipe de stdin do filho
15.      } else {
16.        for (j = 0; j <= i; j++)
17.          close(out[j][READ]);
18.        // redireccionamento do stdout do filho
19.        // redireccionamento do stdin do filho
20.        // execução do comando vecmult
21.      }
22.    }
23.    for (i = 0; i < L; i++) {
24.      char buf[100];
25.      // leitura ordenada do output dos filhos
26.    }
27.  }
```

- Questão I: a linha 14, que corresponde à escrita na pipe de stdin do filho, pode ser implementada pela sequência de instruções:

  a)
```
close(in[READ]);
write(in[WRITE], line, len);
close(in[WRITE]);
```

  b)
```
close(in[WRITE]);
write(in[READ], line, len);
close(in[READ]);
```

  c)
```
close(in[READ]);
fprintf(in[WRITE], "%s", line);
close(in[WRITE]);
```

  d)
```
close(in[WRITE]);
fprintf(in[READ], "%s", line);
close(in[READ]);
```

- Questão II: a linha 18, que corresponde ao redireccionamento do stdout do filho, pode ser implementada pela sequência de instruções:

  a)
```
dup2(out[i][WRITE], 1);
close(out[i][WRITE]);
```

  b)
```
dup2(out[i][WRITE], 0);
close(out[i][WRITE]);
```

  c)
```
dup2(1,out[i][WRITE]);
close(out[i][WRITE]);
```

  d)
```
dup2(0,out[i][WRITE]);
close(out[i][WRITE]);
```

```
-------------------------------------------------------------------------
- Questão III: a linha 19, que corresponde ao redireccionamento do
  stdin do filho, pode ser implementada pela sequência de instruções:

  a) close(in[WRITE]);
     dup2(in[READ], STDIN_FILENO);
     close(in[READ]);

  b) close(in[WRITE]);
     dup2(STDIN_FILENO, in[READ]);
     close(in[READ]);

  c) close(in[READ]);
     dup2(in[WRITE], STDIN_FILENO);
     close(in[WRITE]);

  d) close(in[READ]);
     dup2(STDIN_FILENO, in[WRITE]);
     close(in[WRITE]);

-------------------------------------------------------------------------
- Questão IV: a linha 20, que corresponde à execução do comando
  vecmult, pode ser implementada pela sequência de instruções:

  a) execl("./vecmult", "vecmult", argv[2], NULL);

  b) execl("vecmult", "vecmult", argv[2], NULL);

  c) execl("./vecmult", "vecmult", argv[2]);

  d) execl("vecmult", "vecmult", argv[2]);

-------------------------------------------------------------------------
- Questão V: a linha 25, que corresponde à leitura ordenada do output
  dos filhos, pode ser implementada pela sequência de instruções:

  a) while ((j = read(out[i][READ], buf, 100)) != 0) {
       buf[j] = 0;
       printf("%s", buf);
     }

  b) j = read(out[i][READ], buf, 100);
     buf[j] = 0;
     printf("%s", buf);

  c) while ((j = fscanf(out[i][READ], "%s", buf)) != 0) {
       buf[j] = 0;
```

```
      printf("%s", buf);
    }

  d) fscanf(out[i][READ], "%s", buf);
    printf("%s", buf);
```
