Número: Nome:

LEIC/LETI – 2014/15 - 1º Teste de Sistemas Operativos

22 de Novembro de 2014

Responda no enunciado, apenas no espaço fornecido.
Identifique todas as folhas.
Justifique todas as respostas.
Duração: 1h30m

Grupo I [6 Val]

Considere um dado programa, com vários fios de execução, que tem 3 versões que se executam num computador uniprocessador com sistema operativo do tipo Unix e cuja funcionalidade não implica nenhuma operação de I/O:

- versão V1 em que cada fio de execução é suportado por um processo,
- versão V2 em que cada fio de execução é suportado por uma tarefa real, e
- versão V3 em que cada fio de execução é suportado por uma pseudo-tarefa.

Note que as 3 versões têm exactamente a mesma funcionalidade.

1. [0.5 Val] Como compara as versões V1 e V2 no que respeita à robustez do programa? Na sua resposta considere, por exemplo, que um dos fios de execução faz uma divisão por zero.

2. [0.5 Val] Como compara as versões V1 e V3 no que respeita à robustez do programa? Na sua resposta considere, por exemplo, que um dos fios de execução faz uma divisão por zero.

3. [0.5 Val] Como compara as versões V1 e V2 no que respeita à rapidez de execução do programa?

4. [0.5 Val] Como compara as versões V1 e V3 no que respeita à rapidez de execução do programa?

5. [1 Val] Considere agora que o programa é executado numa máquina com dois processadores. Compare o desempenho das versões V2 e V3.

Considere um processo o seguinte programa que se encontra num ficheiro "t.c" e que é compilado gerado um ficheiro executável "t.exe". Assuma que não ocorre nenhum erro quando o programa se executa.

```
#include <stdio.h>

main() {
   int pid, pid_filho, status;

   printf ("\nantes do fork pid=%d\n", getpid());

   pid = fork();
   if (pid == 0) {
     printf ("pai=%d e filho=%d\n", getppid(), getpid());
     execv ("t.exe",NULL);
   }
   else if (pid != -1){
     pid_filho = wait(&status);
     printf ("depois do wait feito pelo pid=%d obtendo
              pid_filho=%d\n", getpid(), pid_filho);
     exit (0);
   }
   else {
     printf ("erro no fork\n");
     exit (-1);
   }
}
```

6. [1 Val] Quantos processos são criados?

7. [1 Val] Assuma que o fork nunca retorna erro. A instrução depois do "*wait*" é executada ?

8. [1 Val] Assuma que o fork retorna erro na 3ª vez que é chamado. Considere que inicia a execução do programa "t.exe". Indique o seu *output* continuando o que se indica.

antes do fork pid=2204

Grupo II [7 Val]

Considere o seguinte problema:
- existem múltiplas tarefas que por vezes podem querer enviar (produtoras) ou receber (consumidoras) mensagens para/de um canal partilhado;
- o canal tem a capacidade N e é implementado como um *buffer* circular;
- a função `envia` permite enviar múltiplas mensagens para o canal de uma só vez, sendo as mensagens colocadas de forma consecutiva no *buffer* circular; se não houver espaço suficiente, a função `envia` espera até que haja;
- a função `recebe` permite receber a próxima mensagem disponível no canal; se o canal estiver vazio, a função espera até chegar uma mensagem.

Considere a seguinte implementação de uma solução para o problema em causa (em pseudo-código):

```
int canal[N];
int free = N;
int envia_ptr=0, recebe_ptr=0;

envia(Message messages[], int numMsg) {
   while (free < numMsg);
   for each (Message m in messages) {
      canal[envia_ptr] = m;
      envia_ptr = (envia_ptr+1) % N;
      free --;
   }
 }
}
```

```
Message recebe() {
  while (free == N);
  Message m = canal[recebe_ptr];
  recebe_ptr = (recebe_ptr+1) % N;
  free ++;
  return m;
}
```

1. Usando a solução acima apresentada num computador mono-processador, houve utilizadores que detectaram situações anómalas. Para cada situação anómala apresentada abaixo, explique se e porque pode acontecer, ilustrando com um exemplo de execução que produza o mesmo sintoma.

   a. [0.5 Val] "Dois produtores produziram 1 item cada; no entanto, os consumidores apenas encontraram 1 item para consumir."

   b. [0.5 Val] "Um produtor colocou 1 item no *buffer*; no entanto, esse mesmo item foi consumido por 2 consumidores diferentes."

   c. [0.5 Val] "Quando um produtor tenta colocar um item no *buffer*, e este está vazio e há vários consumidores a tentar consumir, o produtor demora muito mais tempo a concluir a operação (do que demoraria se não houvesse consumidores à espera de itens)."

2. Proponha uma solução, em pseudo-código, que elimine ou minimize os problemas apontados acima.
   a. [2 Val] Na sua solução recorra a trincos lógicos e/ou semáforos. No caso dos semáforos, considere que estes suportam as seguintes operações:
      - `esperar (identificador_do_semaforo, numero_de_unidades);`
      - `assinalar (identificador_do_semaforo, numero_de_unidades).`

      Não se esqueça de inicializar todas as variáveis (inclusivamente os semáforos que usar).

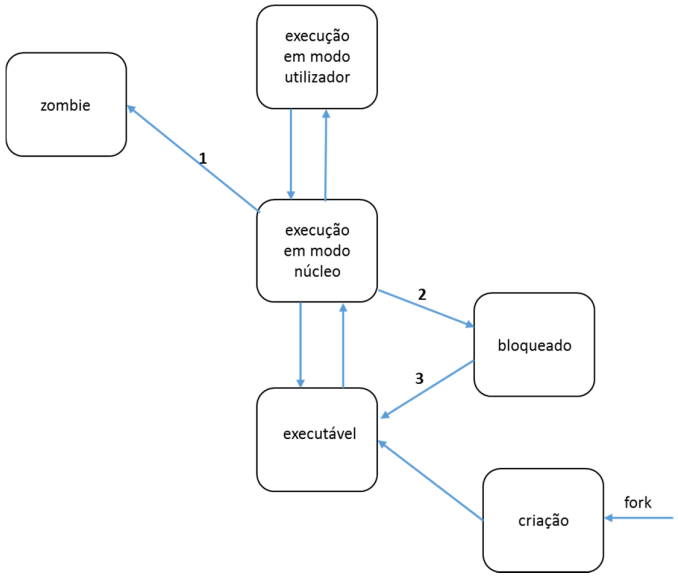
b. [1.5 Val] Explique como é que a solução que apresentou resolve cada um dos comportamentos anómalos apontados acima para a primeira solução (1.a., 1.b, 1.c).

c. [1 Val] Diga se a sua solução permite que mensagens sejam enviadas e recebidas por tarefas distintas que usam índices distintos do vetor permitindo assim o nível máximo de concorrência. Altere a sua solução se necessário.

d. [1 Val] Considere que teria de desenvolver uma solução que utilizasse apenas mecanismos de sincronização indirecta. Qual a desvantagem principal?

Grupo III [7 Val]

Considere o diagrama de estados dos processos em Unix que se apresenta de seguida.



1. [1 Val] Complete o diagrama de estados indicando, para cada uma das transições de estado enumeradas, uma chamada sistema que dispare essa transição; para cada uma dessas transições indique qual o processo que a efectua (i.e., o próprio processo que transita de estado ou um outro).

2. [1 Val] Complete o diagrama de estados, acrescentando um novo estado "suspenso", e represente as transições de estado que são disparadas pelas chamadas "suspender" (*suspend*), "re-activar" ou "acordar" (*resume*), e "adormecer" (*sleep*); para cada uma das transições indique qual o processo que a efectua (i.e., o próprio processo que transita de estado ou um outro processo).

3. [1 Val] Qual a razão para a existência do estado zombie.

4. [0.5 Val] Considere um processo P1 que está no estado "execução em modo utilizador". Diga se é possível que ele transite para o estado zombie directamente, i.e. sem transitar pelo estado "execução em modo núcleo".

5. [1 Val] Considere o sistema de *scheduling* original do Unix e o mecanismo de *scheduling* do Linux baseados em *epochs*. Qual a diferença fundamental entre ambos? O que se ganha/perde com o mecanismo de *epochs*?

6. Considere um sistema operativo do tipo Unix com *time-slice* (também designado por quantum) fixo, prioridades dinâmicas e preempção.

    a. [0.5 Val] Indique uma desvantagem de ter um *time-slice* muito pequeno.

    b. [1 Val] Diga se a política de *scheduling* do Unix privilegia os processos interactivos. Justifique relacionando com a forma de cálculo das prioridades dos processos.

7. [1 Val] Considere agora um sistema operativo cujo escalonador está optimizado para tratamento por lotes. Diga se concorda com a afirmação seguinte, justificando: “Este sistema operativo tem como objectivo dar a cada processo uma fatia equitativa do tempo.”
