Número: Nome:

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

22 de Novembro de 2014

******Versão com soluções.******

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.

   A versão V1 é mais robusta, uma vez que apenas o processo onde ocorre o erro é terminado, continuando as restantes tarefas a executar-se. Na versão V2 o processo (que inclui todas as tarefas) é terminado (ver secção 3.4.2 do livro).

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.

   Resposta idêntica à anterior.

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

   V1 será mais lento, pois a criação de processos e comutação entre processos é mais lenta que a criação/ comutação entre tarefas, dado que os processos não partilham o mesmo contexto.

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

   Resposta idêntica à anterior.

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

   A versão V2 será mais rápida pois diferentes tarefas poderão executar-se concorrentemente nos dois processadores. No versão V3, todas as pseudo-tarefas irão partilhar um único processador.

Considere um processo que executa 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?

Serão criados continuamente processos, de forma recursiva, até que se esgotem os recursos da máquina (ver secção 3.5.2 do livro).

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

Como cada filho nunca retorna ao pai (pois espera pelo seu próprio filho), a instrução wait nunca será executada até que existe um erro (por exemplo, no exec).

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
pai=2204 e filho=2205
antes do fork pid=2205
pai=2205 e filho=2206
antes do fork pid=2206
erro no fork
depois do wait feito pelo pid=2205 e obtendo pid_filho=2206
depois do wait feito pelo pid=2204 e obtendo pid_filho=2205

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):

```
00 Message canal[N];
01 int free = N;
02 int envia_ptr=0, recebe_ptr=0;
03
04 envia(Message messages[], int numMsg) {
05   while (free < numMsg);
06   for each (Message m in messages) {
07      canal[envia_ptr] = m;
08      envia_ptr = (envia_ptr+1) % N;
09      free --;
10   }
11 }
```

```
12 Message recebe() {
13   while (free == N);
14   Message m = canal[recebe_ptr];
15   recebe_ptr = (recebe_ptr+1) % N;
16   free ++;
17   return m;
18 }
```

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."

   Considerem-se dois produtores P1 e P2 a executar "envia" de forma concorrente. P1 perde o processador depois de executar a linha 07 e antes de executar a linha 08. Posteriormente P2 executas essas linhas, escrevendo na mesma posição que P1 (ver secção 6.3.4 do livro).

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

   Exemplo semelhante ao anterior, para dois consumidores e perca de processador entre as linhas 14 e 15.

   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)."

   Sim pois, uma vez que os consumidores executam uma espera activa (linha 13), só quando os consumidores esgotam o quantum é que o produtor consegue obter o CPU.

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).

```
00 Message canal[N];
01 int free = N;
02 int envia_ptr=0, recebe_ptr=0;
pode_enviar = criar_semaforo(N),
pode_receber = criar_semaforo(0);
trinco_enviar = criar_semaforo (1);
trinco_receber = criar_semaforo (1);


03
04 envia(Message messages[], int numMsg) {
05    esperar(pode_enviar, numMsg);
      esperar(trinco_enviar,1);
06    for each (Message m in messages) {
07       canal[envia_ptr] = m;
08       envia_ptr = (envia_ptr+1) % N;
09       free --;
10    }
       assinalar (trinco_enviar,1);
       assinalar (pode_receber, numMesg);
11 }
```

```
12 Message recebe() {
13    esperar(pode_receber,1);
      esperar(trinco_receber,1);
14    Message m = canal[recebe_ptr];
15    recebe_ptr = (recebe_ptr+1) % N;
16    free ++;
      assinalar(trinco_receber,1);
      assinalar(pode_enviar,1);

17    return m;
18 }

/* Nota: para o "trinco_enviar" e
"trinco_receber" poderiam também ser
usados trincos lógicos */
```

   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).

      1.a e 1.b são resolvidos garantindo a exclusão mútua no acessos os índices e 1.c é resolvido evitando as esperas activas.

   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 um maior grau de concorrência. Altere a sua solução se necessário.

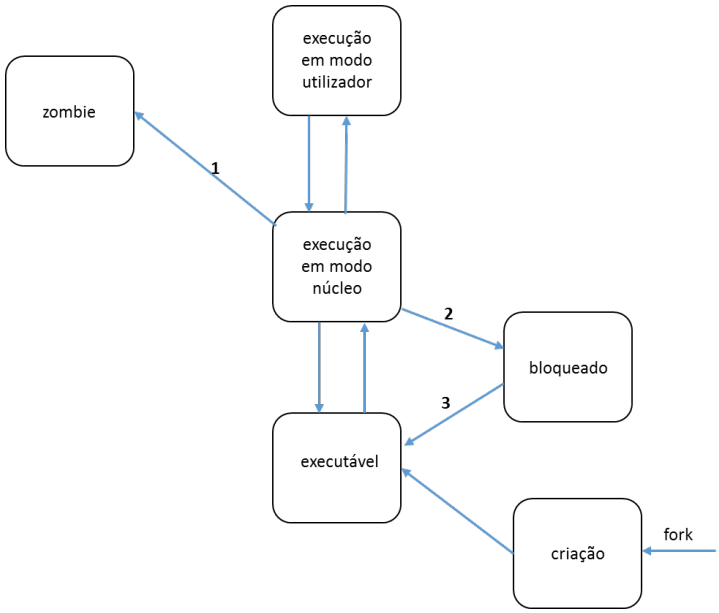
      Sim, uma vez que o trinco que protege o índice de escrita é diferente do trinco que protege o índice de leitura. Este nível de concorrência não poderia ser atingido caso fosse usado um único trinco.

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

      Seria necessário conhecer à partida o número e o identificador de todos os produtores e consumidores, ficando a solução muito menos genérica (ver secção 6.2.2 do livro).

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).

Nota: ilustra-se dando apenas uma de várias chamadas sistema que estimulam as transições (ver secção 4.4.4 do livro).
1) exit pelo próprio processo
2) esperar pelo próprio processo
3) assinalar por outro processo

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).

Ver Figura 4.3 da 2ºa edição do livro da cadeira.

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

É uma maneira expedita de guardar informação sobre um processo que já terminou até que esta seja recolhida pelo pai quando este último faz "wait".

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".

A transição direta é impossível. Terá que fazer exit ou receber uma excepção/ signal.

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*?

O algoritmo de escalonamento do Unix original recalculava as prioridades de todos os processos de forma periódica, independentemente do número de processos e do tempo que estes se tinham

executado desde a última iteração, tonando-se muito pesado para um elevado número de processos. A versão baseada em épocas só recalcula as prioridades depois de todos os processos se terem bloqueado ou terminado o seu quantum, mitigando este efeito. Ver secção 4.4.6 do livro.

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.

As trocas de contexto acontecerão com muito mais frequência, aumentando a fracção de tempo gasto no despacho em relação ao tempo gasto a executar o códigos dos processos.

   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.

Sim, uma vez que as prioridades são uma função do CPU gasto pelos processos (quanto mais CPU um processo gasta, menos prioritário fica). Os processos interactivos passam muito mais tempo bloqueados do que a gastar CPU, acabando por ter mais prioridade.

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.”

Não pois um escalonador para processamento por lotes pretende maximizar o débito do sistema e, desta forma, pretende optimizar o processamento do lote como um todo e não equilibrar tempo de CPU que cada processo do lote individualmente ocupa.