Número: Nome:

LEIC/LETI – 2018/19 - 1º Teste de Sistemas Operativos (Repescagem)

5 de fevereiro de 2019

Responda no enunciado, apenas no espaço fornecido. Identifique todas as folhas.

Duração: 1h30m

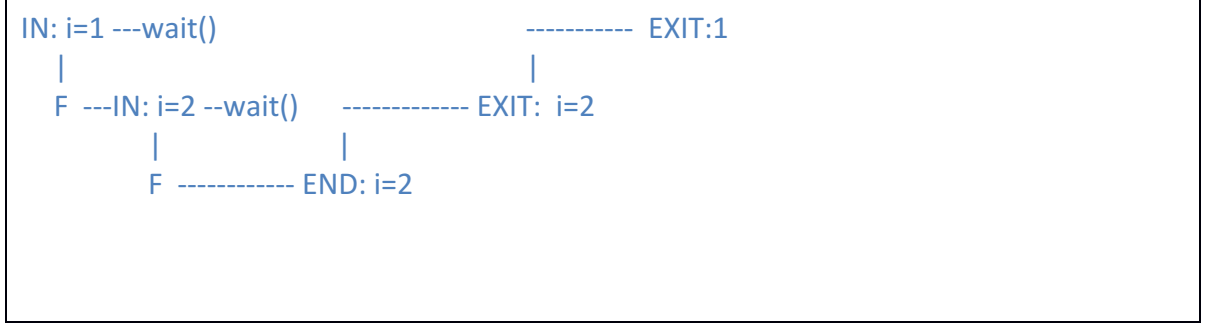
Grupo I [5 Val]

1. Considere o seguinte programa e assuma que as chamadas a fork não falham:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int i=0;

void f() {
    i++;
    printf("IN: i=%d\n",i);
    int pid= fork();
    if (pid!=0) {
        wait(NULL);
        printf("EXIT: i=%d\n",i);
        exit(0);
   }
}

int main() {
   i=0;
   while (i<2)
       f();
   printf("END: i=%d\n",i);
   exit(0);
}
```

   a. [1 val] Quantos processos são criados no total durante uma execução completa deste programa (incluindo o processo inicial)? Justifique a resposta através de uma árvore que ilustre a relação pai/filho entre os vários processos criados pelo programa.

3 processos, incluindo o processo inicial.



   b. [0,5 val] Quantas vezes é imprimido "IN: i=0"? Caso seja uma ou mais vezes, indique qual/quais processo(s) imprimem esta *string*.

0

c. [0,5 val] Quantas vezes é imprimido "IN: i=1"? Caso seja uma ou mais vezes, indique qual/quais processo(s) imprimem esta *string*.

1 vez, pelo processo pai.

d. [1 val] Quantas vezes é imprimido "END: i=2"? Justifique sucintamente a resposta, mencionando qual/quais processo(s) imprimem esta *string*.

1 vez, pelo último processo criado antes de terminar a própria execução da função main.

e. [1 val] Se o programa for executado múltiplas vezes, é possível que o *output* mude? Em caso de resposta negativa ilustre qual é o único *output* possível. Em caso de resposta positiva, ilustre dois exemplos de *outputs* diferentes.

As chamadas à função wait obrigam a ordem de impressão a ser
IN: i=1
IN: i=2
END: i=2
EXIT: i=2
EXIT: i=1

2. [1. val] Que comando se pode utilizar num sistema Linux para detetar a presença de processos *zombie*?

Usando, por exemplo, o comando ps.

Grupo II [9 Val]

Considere uma aplicação paralela que mantém um mapa (mono-dimensional) que consiste num vetor com N posições.
Algumas posições do mapa podem ser preenchidas com *caminhos*. Um caminho consiste numa sequência contígua de posições do mapa ligando um ponto de partida a um ponto de destino.
Cada caminho é identificado por uma letra. Cada posição do caminho tem uma etiqueta numérica, em que a posição inicial do caminho tem o valor 1, a segunda posição do segmento tem o valor 2, e por aí adiante.
Por exemplo, o seguinte mapa foi preenchido por 3 caminhos (a, b e c):

	a,1	a,2	a,3	a,4			b,1	b,2	b,3					c,1	c,2	c,3	c,4	c,5	c,6	

O mapa é partilhado entre múltiplas tarefas (threads) concorrentes, que podem tentar inserir novos caminhos chamando a função *adicionarCaminho*.

1. [1,5 val] Considere a seguinte implementação da função *adicionarCaminho*.

```
typedef struct {
  char id;
  int etiqueta;
} entrada_t;

entrada_t mapa[N]; //inicializado com entradas com id=' '


int adicionarCaminho(unsigned int posInicial, unsigned int posFinal, char id) {
  int i;

  if (posInicial > posFinal || posInicial >= N || posFinal >= N || id == ' ')
    return ERRO;

  for (i = posInicial; i<=posFinal; i++) {
    if (mapa[i].id != ' ')
      return ERRO;
  }

  int etiq=1;
  for (i=posInicial; i<=posFinal; i++) {
    mapa[i].id = id; mapa[i].etiqueta = etiq;
    etiq++;
  }

  return SUCESSO;
}
```

Testou-se esta implementação com múltiplas tarefas que adicionavam caminhos ao mapa. Em algumas execuções, observou-se que o mapa final aparecia corrompido, como por neste exemplo:

	f,1	f,2	g,1	g,2	f,5	f,6													

Descreva uma execução simples que origine este erro.

Duas tarefas, T1 e T2, chamam concorrentemente adicionarCaminho(1,6,'f') e adicionarCaminho(3,4,'g'), respetivamente. T2 ganha primeiro o processador, completa o primeiro ciclo *for* e, antes de executar o segundo ciclo *for*, perde o processador para T1. T1 executa a função completamente, adicionando o seu caminho 'f'. Quando T2 recupera o processador, executa o segundo ciclo *for* assumindo que as posições 3-4 estariam livres, assim escrevendo por cima do caminho 'f'.

2. [3,5 val] Proponha uma solução alternativa que, recorrendo aos mecanismos de sincronização apropriados, corrija o erro acima. A sua solução deve: i) minimizar as alterações ao código acima; ii) permitir paralelismo entre tarefas que estejam a adicionar segmentos disjuntos.

```
mutex_t m[N];

int adicionarCaminho(unsigned int posInicial, unsigned int posFinal, char id) {
  int i;

  if (posInicial > posFinal || posInicial >= N || posFinal >= N || id == ' ')
    return ERRO;

  for (i = posInicial; i<=posFinal; i++) {
    lock(m[i]);
    if (mapa[i].id != ' ') {
      while (i >= posInicial) {
        unlock(m[i]);
        i--;
      }
      return ERRO;
  }

  int etiq=1;
  for (i=posInicial; i<=posFinal; i++) {
    mapa[i].id = id; mapa[i].etiqueta = etiq;
    etiq++;
    unlock(m[i]);
  }

  return SUCESSO;
}
```

3. [1,5 val] Assuma agora que a função *adicionarCaminho* passa também a suportar caminhos com o sentido invertido, ou seja em que *posInicial > posFinal*.
   Descreva sucintamente e objetivamente (em texto, não em código) como estenderia a sua solução anterior para suportar este cenário. A sua proposta deve: i) continuar a permitir paralelismo entre tarefas que adicionam caminhos disjuntos e ii) prevenir interblocagem.

Verificar se *posInicial > posFinal* e, caso sim, construir o caminho na ordem inversa.
Para prevenir interblocagem, fechar os mutexes de cada posição seguindo a ordem crescente dos índices (mesmo no caso de caminhos com sentido invertido).

- [2,5 val] Considere agora a seguinte extensão ao problema:A função *adicionarCaminho* deve passar a assegurar a seguinte condição adicional: *o número máximo de posições que podem estar ocupadas é K (onde K<N).* Ou seja, caso uma tarefa tente inserir um novo caminho sem que a condição acima possa ser satisfeita (ou seja, quando o total de posições já ocupadas mais o número de posições necessário para o novo caminho for superior a K), essa tarefa deve ser bloqueada até a função *limparMapa (*descrita a seguir*)* ter executado.
- Além da função *adicionarCaminho* função, existe uma função *limpaMapa,* que liberta atomicamente todas as posições do mapa.

Assumindo que as funções *adicionarCaminho* e *limparMapa* simples (ou seja, sem a condição de bloqueio indicada acima) já estão corretamente implementadas, implemente as funções *adicionarCaminhoAlt* e *limparMapaAlt* que impõem a condição de bloqueio acima.

Para assegurar a necessária coordenação entre tarefas, a sua solução deve recorrer a *mutexes* e/ou *variáveis de condição* (não pode usar semáforos).

```
//Declarar e inicializar aqui novas variáveis globais que precise
int ocupadas = 0;
mutex_t mOcupadas;
cond_t podeInserir;


int adicionarCaminhoAlt(unsigned int posInicial, unsigned int posFinal, char id) {
  int dim = posFinal - posInicial + 1;

  fechar(mOcupadas);
  while (ocupadas + dim > K)
    wait(podeInserir, mOcupadas);
  ocupadas += dim;
  abrir(mOcupadas);

  adicionarCaminho(posInicial, posFinal, id);

}

int limparMapaAlt() {

  limparMapa();

  fechar(mOcupadas);
  notifyAll(podeInserir);
  ocupadas = 0;
  abrir(mOcupadas);

}
```

Grupo III [6 Val]

1. [1 val] Descreva um cenário onde é necessário utilizar *named pipes* em vez de *pipes* simples (i.e., sem nomes).

Caso os processos não tenham qualquer relação hierárquica.

2. [0,5 val] Ao executar *open(...)* passando como argumento o nome de um *named pipe* é possível que a chamada à função *open* bloqueie? Se sim, até quando?

Sim, até a outra extremidade do pipe não ter sida aberto.

3. [0.5 val] Ao executar *read(...)* passando como argumento o descritor de ficheiro de um *named pipe*, é possível que a chamada à função bloqueie? Se sim, até quando?

Sim, caso o pipe esteja vazio.

4. [0.5 val] Ao executar *write(...)* passando como argumento o descritor de ficheiro de um *named pipe*, é possível que a chamada à função *write* bloqueie? Se sim, até quando?

Sim, caso o pipe esteja cheio.

5. [3,5 val] Suponha que um processo queira aceitar pedidos recebidos através 3 canais: i) um *named pipe*, ii) um *Unix socket stream* e iii) um *INET socket stream*. Assuma que:
   - estes 3 canais já foram apropriadamente inicializados, e que os respetivos descritores de ficheiros são fd1, fd2, fd3.
   - os pedidos têm tamanho fixo de 50 bytes.
   - cada pedido pode ser tratado chamando a função *trataPedido(int fd, char* buf, int size),* onde fd é o descritor de ficheiro a usar para enviar a resposta ao pedido, buf contém o pedido e size é o tamanho do pedido.

Desenvolva a função void *recebePedido(int fd1, int fd2, int fd3)* que, usando apenas uma tarefa:
1. deteta a chegada de novos pedidos através um qualquer dos 3 canais
2. lê os novos pedido e trata-os chamando a função *trataPedido()*

Nota: por simplicidade pode omitir o tratamento de erros

```
void recebePedido(int fd1, int fd2, int fd3) {
   // declaração e inicialização de eventuais variáveis

   fd_set mask, test_mask;
   int selected;
   char buf[50];
   FD_ZERO(&test_mask);
   FD_SET(fd1, &test_mask);    FD_SET(fd2, &test_mask);    FD_SET(fd3, &test_mask);



   for (;;) { // em cada iteração deste ciclo é recebido e tratado um pedido
            mask=testmask
            select(3,&mask,0,0,0);;
            if (FDISSETfd1, &mask)
               selected=fd1;
            else if (FDISSETfd1, &mask)
                selected=fd2;
            else selected=fd3;

            read(selected,buff,50);
            trataPedido(selected, buf)

   } //end for loop
} // end recebePedido
```