Fundamentos de Sistemas de Operação
1º Teste, 5 de Novembro de 2021

QUESTÕES DE ESCOLHA MÚLTIPLA — VERSÃO A

1) Considere a realização de duas funções **`myRCbegin()`** e a sua "dual" ou "complementar", **`myRCend()`**, que se querem baseadas em semáforos e que se destinam, respectivamente, a "marcar" o início e o fim regiões críticas; a "caixa" no meio da figura mostra a inicialização.

```
sem_t mtx;
// L1: inicialização do semáforo
sem_init(&mtx, 0, val);
```

```
void myRCbegin(sem_t *m) {
  func1(m);
}
```

```
void myRCend(sem_t *m) {
  func2(m);
}
```

Qual das alíneas abaixo representa a informação em falta para uma implementação correcta?

a) A *func1* é **`sem_wait`**, a *func2* é **`sem_post`** e *val* é **`0`**
b) A *func1* é **`sem_post`**, a *func2* é **`sem_wait`** e *val* é **`0`**
c) A *func1* é **`sem_post`**, a *func2* é **`sem_wait`** e *val* é **`1`**
d) A *func1* é **`sem_wait`**, a *func2* é **`sem_post`** e *val* é **`1`**
e) A *func1* é **`sem_wait`**, a *func2* é **`sem_post`** e *val* é **`PTHREAD_MUTEX_INITIALIZER`**

2) O espaço de endereçamento de um processo é:

a) o conjunto de todas as posições de memória real
b) o conjunto das posições de memória virtual que potencialmente podem ser acedidas pelo processo durante a sua execução
c) o conjunto de posições de memória virtual que constituem as zonas de código, *stack* e *heap*
d) o conjunto das posições de memória física que podem ser acedidas pelo processo durante a sua execução
e) o conjunto de todas as posições de memória física, do menor ao maior endereço físico

3) O objetivo **fundamental** de um sistema de operação é:

a) isolar entre si os utilizadores e os seus processos
b) suportar a execução de aplicações
c) permitir aos utilizadores aceder com facilidade ao hardware
d) permitir aos programadores aceder com facilidade ao hardware
e) permitir aos administradores de sistema gerir com facilidade o hardware

4) A característica **fundamental** de um sistema de operação que suporta multiprogramação é permitir:

a) controlar de forma concorrente ("simultânea") todo o hardware
b) executar de forma concorrente ("simultânea") vários processos
c) executar de forma concorrente ("simultânea") várias *threads*
d) controlar de forma concorrente ("simultânea") vários periféricos
e) controlar de forma concorrente ("simultânea") vários CPUs

5) Considere as três realizações, "A", "B" e "C", de uma função **`contar()`** que é lançada por duas *threads* distintas e conta o número total de iterações realizado pelo conjunto das duas *threads*, deixando o resultado final em **`res`**. O semáforo **`s`** só é utilizado na função contar "B"; considere-o inicializado a zero. Da mesma forma considere o mutex **`m`** inicializado.

```
#define MAX 100
int res= 0;
sem_t s; pthread_mutex_t m;

// Declaração da função contar

int main(...) {
  pthread_t p1, p2;
  pthread_create(&p1, NULL, contar, NULL);
  pthread_create(&p2, NULL, contar, NULL);
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);

  // Só quando usar a função B
  // sem_getvalue(&s, &res);

  return 0;
}
```

Nota: a função **`sem_getvalue()`** permite obter o valor inteiro "armazenado" num semáforo

```
void *contar(void *a) {
  int i, cnt= 0;
  for (i=0; i<MAX; i++)
    cnt++;
  pthread_mutex_lock(&m);
    res= res + cnt;
  pthread_mutex_unlock(&m);
  return NULL;
}
```
A

```
void *contar(void *a) {
  sem_post(s);
}
```
B

```
void *contar(void *a) {
int i;
  for (i=0; i<MAX; i++) {
    pthread_mutex_lock(&m);
      res++;
    pthread_mutex_unlock(&m);
  }
}
```
C

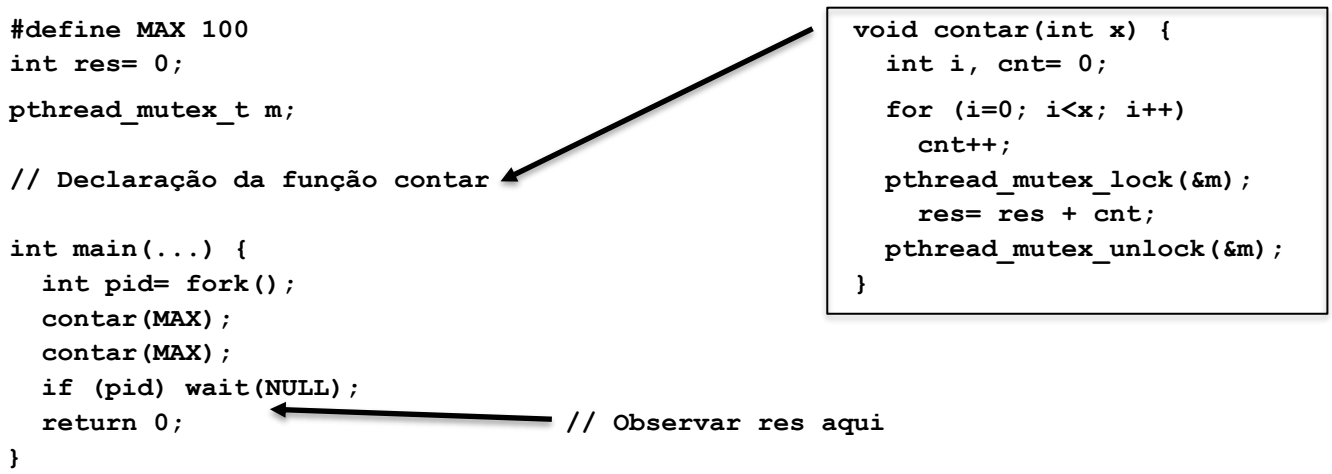
Qual das seguintes afirmações é verdadeira sobre o valor em **`res`** (imediatamente antes do **`return 0`**)?

a) As três funções produzem resultados incorrectos, pois nenhuma respeita a exclusão mútua
b) As três funções produzem resultados correctos, porque ou respeitam a exclusão mútua ou realizam operações "atómicas"
c) Apenas as funções "A" e "C" produzem resultados correctos, porque só elas respeitam a exclusão mútua
d) Apenas a função "C" produz um resultado correcto, porque só ela respeita a exclusão mútua
e) Apenas as funções "A" e "B" produzem resultados correctos, porque ou respeitam a exclusão mútua ou realizam operações "atómicas"

6) A transformação de um programa fonte (em C) num programa executável (linguagem máquina) envolve sucessivas transformações realizadas por vários utilitários. Indique a sequência em que ocorrem essas transformações. **Nota:** sempre que possível será indicada a função desempenhada pelo utilitário, seguida do nome do mesmo entre parêntesis e usando a fonte Courier-Bold; se a descrição da função for demasiado longa, limitamo-nos a indicar o nome do utilitário.

a) pré-processador C (**`cpp`**), compilador (**`cc`** ou **`gcc`**), assembler (**`as`**), ligador/*linker* (**`ld`**)
b) pré-processador C (**`cpp`**), compilador (**`cc`** ou **`gcc`**), assembler (**`as`**), ligador/*linker* (**`ld`**), **`objdump`**
c) compilador (**`cc`** ou **`gcc`**), pré-processador C (**`cpp`**), assembler (**`as`**), ligador/*linker* (**`ld`**)
d) pré-processador C (**`cpp`**), compilador (**`cc`** ou **`gcc`**), assembler (**`as`**), **`objdump`**
e) compilador (**`cc`** ou **`gcc`**), pré-processador C (**`cpp`**), assembler (**`as`**), ligador/*linker* (**`ld`**), **`objdump`**

7) Considere a função **`contar()`** que é lançada no programa principal (listado à esquerda da figura) deixando o resultado final em **`res`**. Considere **o mutex `m` partilhado entre os processos** e inicializado.



```
#define MAX 100
int res= 0;
pthread_mutex_t m;

// Declaração da função contar

int main(...) {
  int pid= fork();
  contar(MAX);
  contar(MAX);
  if (pid) wait(NULL);
  return 0;
}
```

```
void contar(int x) {
  int i, cnt= 0;
  for (i=0; i<x; i++)
    cnt++;
  pthread_mutex_lock(&m);
    res= res + cnt;
  pthread_mutex_unlock(&m);
}
```

// Observar res aqui

Qual das seguintes afirmações é verdadeira sobre o valor em **`res`** (imediatamente antes do **`return 0`**)?

a) O valor de **`res`** é **`400`** pois **`contar`** é executada quatro vezes
b) O valor de **`res`** é **`200`** pois apesar de **`contar`** ser executada quatro vezes, as contagens efectuadas pelo filho re-escrevem ("por cima") os valores guardados no pai
c) O valor de **`res`** é **`indeterminado`** pois apesar de **`contar`** ser executada quatro vezes, as contagens efectuadas pelo filho re-escrevem ("por cima") os valores guardados no pai
d) O pai não termina, fica preso no **`wait()`** (nunca chega ao ponto para observar **`res`**)
e) O valor de **`res`** é **`200`** pois apesar de **`contar`** ser executada quatro vezes, a variável **`res`** vista pelo processo pai é distinta da variável **`res`** vista pelo processo filho

8) Considere um SO concebido para ser **usado num *laptop* moderno** (e.g., Windows 10, MacOS X, Ubuntu Desktop), com os periféricos típicos que este tipo de computadores tem. Que características deve ter um tal SO? [**Assinale todas as opções verdadeiras**]

a) Só suportar/funcionar com periféricos (e.g., discos) validados pelo "fabricante" do SO
b) Adaptar-se sem necessidade de reconfiguração a mudanças de hardware (mais ou menos RAM)
c) Privilegiar a execução de programas interactivos sobre os não-interactivos
d) Executar as operações sobre periféricos rápidos usando espera activa
e) Nenhuma das outras opções está correta

9) Considere um SO concebido para ser **usado num *laptop* moderno** (e.g., Windows 10, MacOS X, Ubuntu Desktop). Qual é a política que deve, **fundamentalmente**, ditar as acções do escalonador nesse SO?

a) FIFO
b) SJF
c) Uma que maximize o *throughput* (débito) dos *jobs* (trabalhos)
d) Uma que favoreça o tempo de resposta
e) Nenhuma das outras opções está correta

10) Considere um programa, escrito originalmente em *assembly*, que inclui uma instrução **`INT 0x80`** – *Software Interrupt* nº 80_{16} – também referida como instrução *trap*); diga o que acontece quando na *shell* o utilizador executa o referido programa e **a execução chega a essa instrução**:

a) se (e só se) o processo corre como *root*, comuta para modo supervisor, executa o INT, e continua
b) se (e só se) o processo corre como *root*, executa o INT, comuta para modo supervisor, e chama o SO
c) executa o INT, comuta para modo supervisor, e chama o SO
d) o processo é abortado pelo SO
e) a execução nunca chega à instrução **`INT`**; o programa é logo abortado, mal é lançado

13) Qual é o número de processos **novos/criados** (excluindo, portanto, o processo inicial) quando um processo executa duas instruções **`fork()`** em sequência, i.e.,

a) 1
b) 2
c) 3
d) 4
e) Erro: não é permitido executar *forks* seguidos sem um **`if`** em cada um

```
...
fork();
fork();
...
```

14) Qual o número de **novas** *threads* **activas** no ponto de observação **`<Ponto O>`** quando um processo executa a sequência de instruções na "caixa"

a) 0
b) 1
c) 2
d) Programa errado; a ordem dos *joins* está trocada
e) Programa errado, a função **`f`** não pode ser lançada duas vezes

```
pthread_create(&t,NULL,f,NULL);
pthread_create(&q,NULL,f,NULL);
pthread_join(&q,NULL);
pthread_join(&t,NULL);
<Ponto O>
```

15) No núcleo do sistema de operação, quem é responsável por desencadear as acções **fundamentais** necessárias à comutação de processos (*process switching*)?

a) O *driver* de CPU
b) O interpretador de comandos (CLI/shell) aquando da execução de um novo processo
c) O módulo de gestão de memória em conjunto com o *driver* de CPU
d) O módulo de gestão de ficheiros, em conjunto com o módulo de gestão de memória
e) Nenhuma das opções anteriores está correta

escalonador

16) Considere dois processos que partilham um *pipe*, sendo que um deles apenas lê do *pipe* (**não** tendo fechado o descritor de escrita) e o outro apenas escreve (**não** tendo fechado o descritor de leitura). O processo escritor já não pretende enviar mais dados e fecha o descritor de escrita; que acontece se o leitor executar um **`read()`**?

a) o leitor bloqueia
b) o programa aborta com um erro "*broken pipe*"
c) ambos, escritor e leitor, bloqueiam
d) o programa aborta com um erro "*segmentation fault*"
e) o leitor recebe como retorno do **`read()`** o valor 0

17) O mapa de memória (MM) de um processo é criado a partir da versão executável do programa (o ficheiro). Contudo, **nem todas** as regiões existentes no MM do processo **têm uma correspondência "perfeita"** com as diferentes zonas do executável. Em que alínea abaixo estão referidas duas dessas zonas?

a) Código (*text*) e Constantes
b) Variáveis Inicializadas (*data*) e Variáveis Não-Inicializadas (`.bss`, ou *Block Started by Symbol*)
c) Código (*text*) e *Stack*
d) *Stack* e *Heap*
e) Variáveis Inicializadas (*data*) e *Stack*

18) O conceito de processo, e a sua realização, permitem explorar os múltiplos CPUs, ou *cores*, dos computadores modernos. Não é **desnecessário** os SOs também suportarem *threads*?

a) Não, porque é mais rápido lançar *threads* do que processos.
b) Não, porque os processos são (por omissão) entidades isoladas, e há problemas cuja solução requer muita partilha de informação
c) Não, porque a comutação entre *threads* de um mesmo processo é mais rápida que a comutação entre processos
d) Todas as anteriores são razões válidas para um SO suportar *threads*
e) Algumas das anteriores não justificam o suporte de *threads*; bastava ter apenas processos

19) Como é implementado o mecanismo de fatias de tempo usado para escalonar processos/*threads*?

a) O SO interrompe periodicamente o CPU
b) Existe um dispositivo *hardware* que interrompe periodicamente o CPU
c) Há uma *thread* do escalonador que interrompe periodicamente o CPU
d) Recorrendo às interrupções geradas pelos diferentes periféricos (discos, placa de rede, gráfica, etc.) que quando em funcionamento, usam interrupções para pedir serviços ao SO
e) Cada vez que um periférico gera uma interrupção

20) Considere o programa abaixo listado no qual as linhas estão numeradas. Que linhas, quando o correspondente código máquina é executado, requerem uma chamada ao SO para completar a sua execução?

```
1  int main(int argc, char *argv[])
   {
2    int i;
3    i = atoi(argv[1]); // função de biblioteca que converte uma
                        // string num inteiro

4    for (k= 0; k < i; k++)
5      printf("%d\n", i);

6     exit(0);
   }
```

a) 5 e 6.
b) 3, 5 e 6
c) 2, e 3
d) 2, 5 e 6
e) 2, 3, 5 e 6