Fundamentos de Sistemas de Operação
1º Teste, 23 de Outubro de 2019

NOME DO ESTUDANTE: _____ Nº: _____

A duração do teste é 1h45 incluindo a tolerância. Nas perguntas de escolha múltipla, as respostas erradas descontam, e a pergunta pode acabar por ter uma classificação negativa, que pode DESCONTAR até 25% da classificação da mesma.

Transcreva para esta caixa as letras que indicam as opções que escolheu em cada uma das perguntas de escolha múltipla. SÓ A SUA RESPOSTA NESTA CAIXA SERÁ CONSIDERADA.

VERSÃO DO TESTE: A (copiar do enunciado)

Cotações aproximadas: escolha múltipla, total de 7 valores; desenvolvimento, total de 13 valores

---

Fundamentos de Sistemas de Operação
1º Teste, 23 de Outubro de 2019

QUESTÕES DE ESCOLHA MÚLTIPLA — VERSÃO A

1) O espaço de endereçamento de um processo é:
a) o conjunto de posições de memória virtual
b) o conjunto de posições de memória que podem ser acedidas pelo processo durante a sua execução.
c) o conjunto de posições de memória que constituem as zonas de código, *stack* e *heap*
d) o conjunto de posições de memória física

2) O objectivo fundamental de um sistema de operação é:
a) suportar a execução de aplicações
b) permitir aos utilizadores aceder com facilidade ao hardware
c) permitir aos programadores aceder com facilidade ao hardware
d) gerir o hardware

3) Um sistema de operação que suporta multiprogramação permite:
a) controlar de forma concorrente ("simultânea") todo o hardware
b) executar de forma concorrente ("simultânea") vários processos
c) controlar de forma concorrente ("simultânea") vários periféricos
d) controlar de forma concorrente ("simultânea") vários CPUs

4) Uma instrução (máquina) privilegiada só pode ser executada quando:
a) o processo em execução corre com privilégio *root* e gera uma interrupção
b) o CPU está no modo de execução supervisor
c) o processo em execução corre com privilégio *root*
d) o processo em execução corre com privilégio *root* e recebe uma interrupção

5) No escalonamento de processos desencadeado por I/O, o escalonador substitui o processo em execução por um outro quando:
a) termina a fatia de tempo (*timeslice*)
b) o processo em execução pede um I/O
c) termina o I/O anteriormente pedido por um processo
d) um processo mais prioritário fica pronto (READY) para execução

6) Num escalonador funcionando por preempção (fatias de tempo) quando, findo o timeslice, um processo A (em execução) é retirado do CPU para "dar lugar" a um outro, B:: [Nota: `RUNNING` = em execução, `READY` = pronto, `WAITING` ou `BLOCKED` = em espera]
a) A transita do estado `RUNNING` para `WAITING` e B do estado `READY` para `RUNNING`
b) A transita do estado `RUNNING` para `READY` e B do estado `READY` para `RUNNING`
c) A transita do estado `RUNNING` para `WAITING` e B do estado `WAITING` para `RUNNING`
d) A transita do estado `RUNNING` para `READY` e B do estado `WAITING` para `RUNNING`

7) Uma troca de contexto (*context switch*) é:
Também aceite como certa
a) a substituição de um processo em execução por outro, na sequência de um acto de escalonamento
b) a transição da execução de um processo do modo utilizador para o modo supervisor
c) a substituição de um processo em execução por outro, na sequência de uma interrupção
d) a substituição de um processo em execução por outro, no fim de uma fatia de tempo

8) Seja o endereço virtual (ou lógico) 1234_{10} == $4d2_{16}$ == 0100 1101 0010_2 numa arquitectura com bus de endereços de 16 bits e páginas de 512_{10} == 200_{16} == 0010 0000 0000_2 bytes. O endereço referido corresponde a:

a) página 2 e deslocamento 210_{10} == $d2_{16}$ == 0 1101 0010_2
b) página 4 e deslocamento 210_{10} == $d2_{16}$ == 0 1101 0010_2
c) página 1 e deslocamento 210_{10} == $d2_{16}$ == 0 1101 0010_2
d) página 9 == 1001 e deslocamento 82_{10} == 52_{16} == 0 0101 0010_2

9) Num sistema com suporte para paginação (mas sem suporte a paginação-a-pedido, matéria que não vem para este teste) uma tabela de páginas (*page table*) indica:

a) para os processos que partilhem memória, em que *frames* da memória estão colocadas as páginas partilhadas desses processos
b) para uma thread, em que *frames* da memória estão colocadas as páginas dessa thread
c) para um processo, em que *frames* da memória (RAM) estão colocadas as páginas desse processo
d) para um processo, em que páginas da memória estão colocadas as *frames* desse processo

10) Qual o número de processos novos/criados (excluindo, portanto, o processo que chama as funções) quando um processo executa duas instruções **`fork()`** em sequência, i.e.,

a) 5
b) 2
c) 3
d) 4

```
…
fork();
fork();
…
```

11) Qual o número de threads novas/criadas (excluindo, portanto, a thread que chama as funções) quando um processo executa duas instruções **`pthread_create()`** em sequência, mas em que as duas chamadas usam o código de uma mesma função, **`f()`**, i.e.,

a) 2
b) 3
c) 4
d) 5

```
…
pthread_create(…,NULL,f,NULL);
pthread_create(…,NULL,f,NULL);
…
```

12) Considere as threads T1 e T2 concorrentemente executadas e duas variáveis partilhadas devidamente inicializadas com o valor 0, sendo **`c`** um inteiro e **`s`** um semáforo inicializado a 1. Qual é o resultado final que se obtém em **`c`** após execução das instruções inscritas nas "caixas":

```
T1
…
sem_wait(&s);
   c--;
sem_post(&s);
   c--;
…
```

```
T2
…
sem_post(&s);
   c++;
sem_post(&s);
   c++;
…
```

a) o valor de c pode não ser observável porque as threads podem bloquear-se mutuamente (*deadlock*)
b) o valor de c é 0
c) o valor de **`c`** pode variar de execução para execução
d) o POSIX diz que é incorrecto fazer dois **`sem_post`** sucessivos para o mesmo semáforo

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

a) bloqueia
b) sofre um *segmentation fault*
c) recebe como retorno do **`read()`** o valor 0
d) recebe como retorno do **`read()`** o valor -1 (e pode obter o erro na variável **`errno`**)

14) Não é possível oferecer um mecanismo de trincos (*locks*) genéricos sem nenhum tipo de suporte (ou auxílio) por parte do sistema de operação porque:

a) as instruções máquina necessárias são privilegiadas
b) nesse caso, as soluções seriam sempre do tipo espera activa (*busy waiting* ou *spin waiting*)
c) uma tal solução iria interferir com o escalonador
d) as instruções máquina necessárias não existem em alguns CPUs modernos

Fundamentos de Sistemas de Operação
1º Teste, 23 de Outubro de 2019

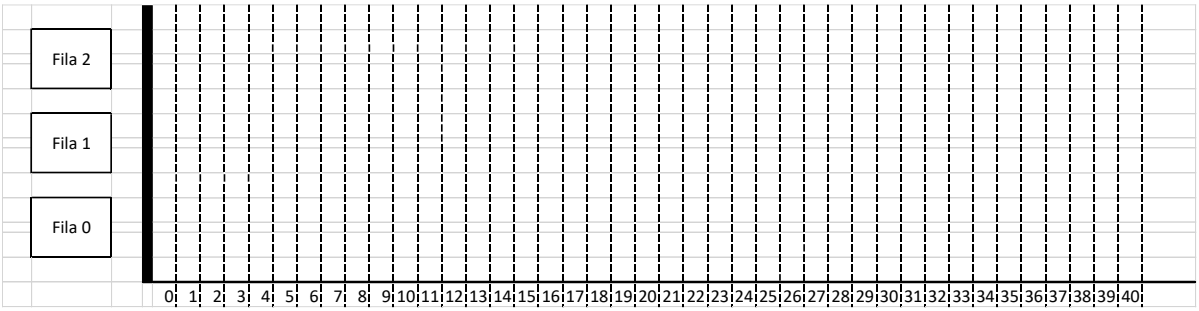
QUESTÕES DE DESENVOLVIMENTO — VERSÃO A

D1) Considere um escalonador MLFQ com 3 filas, de prioridade crescente, Fila 0, Fila 1 e Fila 2. O *timeslice* é de 10 ms, e é igual em todas as filas. O crédito (*allotment*) é de 10ms. Chegam ao sistema 3 processos (jobs) praticamente ao mesmo tempo, com as seguintes características:

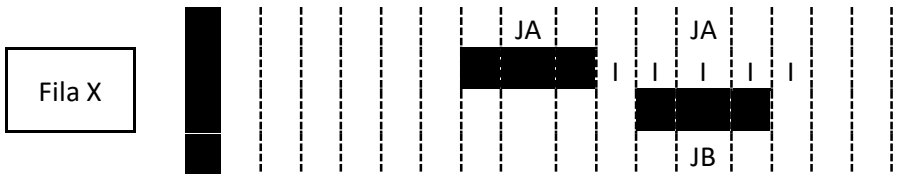
J0: para executar até ao fim, precisa de usar 17ms de CPU. Faz uma operação de I/O a cada 7ms
J1: para executar até ao fim, precisa de usar 8ms de CPU. Faz uma operação de I/O a cada 3ms
J2: para executar até ao fim, precisa de usar 10ms de CPU. Faz uma operação de I/O a cada 4ms

Cada I/O tem a duração de 5ms. O escalonador não usa aumento (*boost*) de prioridade.

Desenhe a evolução temporal (*timeline*) dos 3 jobs.



Sugestão para preenchimento: abaixo, marcaram-se a negro as "caixas" em que o Job A (JA) está a usar o CPU, e com a letra I as caixas em que o JA está a fazer I/O. Para não haver dúvidas colocou-se acima e abaixo das zonas "pintadas" a identificação do Job (neste caso, JA).



D2) Considere as variáveis globais, o programa `main()` (do qual existem 2 versões) e a função `myThread()` abaixo listados. Considere ainda que o sistema onde é executado o programa tem vários CPUs (ou vários *cores*).

```
volatile int contador = 0; // variável global

void *myThread(void *arg) {
  for (int i = 0; i < NUMVEZES; i++)
    contador = contador + 1
  return NULL;
}
```

```
// Versão A
int main(int argc, char *argv[]) {
  pthread_t p1, p2;

  pthread_create(&p1, NULL, mythread, NULL);
  pthread_create(&p2, NULL, mythread, NULL);

  pthread_join(p1, NULL);
  pthread_join(p2, NULL);

  return 0;
}
```

```
// Versão B
int main(int argc, char *argv[]) {
  pthread_t p1, p2;

  pthread_create(&p1, NULL, mythread, NULL);
  pthread_join(p1, NULL);

  pthread_create(&p2, NULL, mythread, NULL);
  pthread_join(p2, NULL);

  return 0;
}
```

a) Considere a Versão A do programa `main()`. Acha que o resultado obtido no `contador` no final da execução (depois do ciclo ser efectuado `NUMVEZES` por cada *thread*) está correcto? Justifique.

Pode, mas na maioria das vezes não estará. Razão: **`contador = contador + 1`** pode corresponder a 3 instruções máquina – 1) mov contador,reg; 2) inc reg; 3) mov reg,contador. Uma thread pode ter acabado 2) e outra ser escalonada; nesse caso o valor de reg é preservado no TCB, mas a nova thread estará a incrementar um valor anterior: perdeu-se um incremento. Isto é designado *race*.

b) Considere a Versão B do programa `main()`. Acha que o resultado obtido no `contador` no final da execução (depois do ciclo ser efectuado `NUMVEZES` por cada *thread*) está correcto? Justifique.

Sim; a 2ª thread só é lançada quando a 1ª acaba de incrementar o contador e termina. Não há acesso concorrente à variável por parte das threads.

c) Se considerou que os resultados obtidos em a) ou b) estão incorrectos, corrija-os sem alterar o programa `main()`; pode alterar a declaração de variáveis, e/ou a função. A sua implementação tem de permitir utilizar mais do que um CPU ou *core* do sistema. **Indique se a sua implementação se destina a corrigir a alínea a) ou b)**.

Declaração de variáveis:

A corrigir a alínea: a)

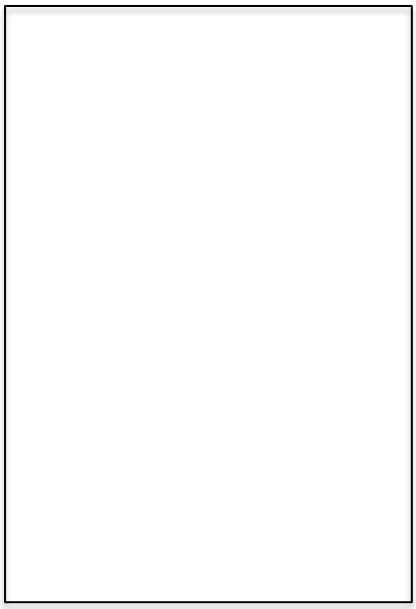
```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

Nova função `myThread`:

```
volatile int contador = 0; // variável global

void *myThread(void *arg) {
  for (int i = 0; i < NUMVEZES; i++)
    pthread_mutext_lock(&m);
      contador = contador + 1;
    pthread_mutext_unlock(&m);
  return NULL;
}
```

D3) Considere o programa listado abaixo, à direita, e o rectângulo à esquerda.



```
const char strC[]= "Teste de FSO\n";
char strV[]= "Teste de FSO\n";

int gV; int gVI= 1;

int main(int argc, char *argv[]) {
  char *ptr; int x;

  printf("main: %p\n", main);
  printf("strC: %p\n", strC);
  printf("strV: %p\n", strV);
  printf("gVI: %p\n", &gVI);
  printf("gV: %p\n", &gV);
  x = 3; printf("x: %p\n", &x);

  ptr = malloc(100e6);
  printf("c)?: %p\n", &ptr);
  printf("c)??: %p\n", ptr);

  //d)?
  ptr= strC; // equivale a ptr= &strC[0]
  *ptr= 'A'; // equivale a strC[0]= 'A'

  return 0;
}
```

O rectângulo vai servir para desenhar o mapa de memória (ou mapa do espaço de endereçamento - EE) do processo que corresponde ao programa listado, quando em execução.

a) O programa é idêntico a um que foi usado nas aulas teóricas e práticas. Imagine que executava concorrentemente (em janelas distintas) várias cópias do programa: o que é que os resultados obtidos nos permitem concluir acerca da arquitectura hardware e sistema de operação da máquina onde está a correr?

Os vários endereços mostrados pelos prints seriam exactamente iguais, o que só é possível se eles forem endereços virtuais. Logo, a arquitectura hardware suporta tradução de endereços e o SO usa uma técnica de Memória Virtual

b) Desenhe (no rectângulo à esquerda do programa) as diversas regiões do EE do processo, identificando-as com “código”, dados, etc. Em sequida, dentro de cada região coloque as “entidades” ou “objectos” do programa que acha que vão “existir” nessa região.

c) Considere as instruções assinaladas com c)? e c)??

O que nos mostra a primeira?

O endereço de ptr, que fica na zona de stack

E o que nos mostra a segunda?

O endereço apontado por ptr, que fica no heap

d) Observe agora as duas instruções que se seguem à zona assinalada com d) no fim do programa. O que acontece quando se executa a última instrução (isto é, `*ptr= 'A';`). Porquê?

Há uma tentativa de violação das permissões da página onde está armazenada a string strC, que por ter sido declarada como constante é colocada numa página que não tem a permissão (w)rite. Isto causa um segmentation fault, e o programa aborta.

D4) Implemente a seguinte função que cria um *pipe* e que redireciona o standard input do processo para a entrada de leitura desse pipe.A função deve retornar 0 se o redireccionamento foi realizado com sucesso ou -1, no caso contrário.

```
int redirInputParaPipe() {
  int fd[2];
  if (pipe(fd) < 0)  return -1;
  if (dup2(fd[0], 0) < 0) return -1;
   // ALTERNATIVA:
  // if ( close(0) < 0) return -1;
  // dup(fd[0]);
  return 0;
}
```

D5) Considere três threads, T1, T2 e T3, prontos para executar os seguintes códigos:

T1:	T2:	T3:

`printf("F");`
`sem_post(&sem1);`
`sem_wait(&sem1);`
`printf (":");`
 	
`printf(")");`
`sem_post(&sem1);`
	`sem_wait(&sem1);`
	`printf("S");`
	`sem_post(&sem2);`
	`sem_wait(&sem1);`
	`printf (":");`
	
	`printf(")");`
	`sem_post(&sem1);`
		`sem_wait(&sem2);`
		`printf("O");`
		
		`printf (":");`
		
		`printf(")");`
		`sem_post(&sem1);`

Utilize semáforos para sincronizar os *threads* por forma a que o resultado da sua execução seja sempre:

```
FSO:):):)
```

Recorra à caixa seguinte para declarar e inicializar os semáforos necessários e complete (escrevendo nos espaços vazios deixados em T1, T2 e T3) o código dos *threads* com as operações de sincronização,

```
sem_t  sem1, sem2;

sem_init(&sem1, 0, 0);
sem_init(&sem2, 0, 0);
```

Protótipos de funções, declarações de tipos, e outras informações úteis
(nota: simplificadas de acordo com a forma de utilização em FSO)

```
pthread_mutex_t mut
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER
pthread_mutex_lock(pthread_mutex_t *mut)
pthread_mutex_unlock(pthread_mutex_t *mut)
pthread_create(pthread_t *thr, NULL, void *(*func) (void *), void *arg);
pthread_join(pthread_t *thr, void **ret)


sem_t var
sem_init(sem_t *var, 0, valor)
sem_post(sem_t *var)
sem_wait(sem_t *var)
```