Fundamentos de Sistemas de Operação
1º Teste, 7 de Novembro de 2020

NOME DO ESTUDANTE: _____ Nº: _____

A duração do teste é 1h45 (incluindo a tolerância). Nas questões de escolha múltipla, as respostas erradas têm uma cotação negativa correspondente a 20% da classificação da questão. Por exemplo, se errar a resposta a uma questão cotada para 1.0 valor, terá uma cotação de –0,2 valores nessa questão. O calssificação total das questões de escolha múltipla pode, portanto, ser negativa.

As questões de escolha múltipla devem ser respondidas na folha própria para o efeito.
Para anular uma resposta coloque uma cruz por cima e pinte a nova resposta (✗ ○ ○ ● ○).
Para reativar uma resposta previamente anulada faça um círculo à volta da resposta a reativar (◉ ○ ○ ✗ ○).

As questões de desenvolvimento devem ser respondidas no próprio enunciado.

Fundamentos de Sistemas de Operação
1º Teste, 7 de Novembro de 2020

QUESTÕES DE ESCOLHA MÚLTIPLA — VERSÃO A

1) Numa arquitetura Intel x86, a dimensão máxima possível para o espaço de endereçamento de um processo é determinada:

a) pela dimensão dos registos genéricos (eax, ebx, ecx, …) do CPU
b) pela dimensão dos registos de endereçamento (esp, ebp, eip) do CPU
c) pela dimensão dos registos do *Translate Lookaside Buffer* (TLB)
d) pela dimensão da memória física instalada
e) pela capacidade da cache do *Translate Lookaside Buffer* (TLB)

2) Um TLB (T*ranslation Lookaside Buffer*) é, na sua essência:

a) um dispositivo *hardware*, logicamente colocado entre o CPU e a RAM, que reserva uma porção de RAM para servir como *buffer*, e que é usado para acelerar a tradução de endereços virtuais em reais
b) um módulo do SO que recorre a uma área de *buffer* em RAM para acelerar a tradução de endereços virtuais em reais
c) um dispositivo *hardware*, colocado entre o CPU e a cache nível L0, que usa o conteúdo desta cache L0 para acelerar a tradução de endereços virtuais em reais
d) um dispositivo *hardware*, logicamente colocado entre o CPU e a RAM, que contém um *buffer* interno próprio e é usado para acelerar a tradução de endereços virtuais em reais
e) um dispositivo *hardware*, logicamente colocado entre os níveis de *cache* L0 e L1, e é usado para acelerar a tradução de endereços virtuais em reais

3) 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

4) 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

5) 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) executar de forma concorrente (“simultânea”) várias threads
d) controlar de forma concorrente (“simultânea”) vários periféricos
e) controlar de forma concorrente (“simultânea”) vários CPUs

6) Considere um programa, escrito originalmente em *assembly*, que inclui uma instrução privilegiada (p. ex. a instrução **`CLI`** – CLear Interrupts – o CPU deixa de atender interrupções); diga o que acontece quando na shell o utilizador executa o programa e a execução chega a essa instrução:

a) se o processo corre com privilégio *root*, comuta para modo supervisor, executa o CLI, e continua
b) se o processo corre com privilégio *root*, executa o CLI, comuta para modo supervisor, e chama o SO
c) se o processo corre sem privilégio *root*, aborta com *stack overflow*
d) o processo é abortado pelo SO
e) a execução nunca chega à instrução **`CLI`**; o programa é logo abortado mal é lançado

7) Num SO multiprogramação, a gestão de memória (GM) por **paginação**, quando comparada com a GM por **partições fixas**, [Nota: EE = Espaço de Endereçamento]

a) permite um melhor aproveitamento do espaço (na RAM), mas não só a velocidade de acesso à RAM é menor como apenas permite proteger o EE do processo como um todo, e não cada zona (código, dados, *stack*, *heap*) individualmente
b) permite um melhor aproveitamento do espaço (na RAM) e uma proteção individual das diferentes zonas do EE do processo (código, dados, *stack*, *heap*), mas a velocidade de acesso à RAM é menor
c) permite uma proteção individual das diferentes zonas do EE do processo (código, dados, *stack*, *heap*), mas não só a velocidade de acesso à RAM é menor como há um pior aproveitamento do espaço na RAM, como resultado da fragmentação interna
d) permite um melhor aproveitamento do espaço na RAM, uma proteção individual das diferentes zonas do EE do processo (código, dados, *stack*, *heap*) e uma maior velocidade de acesso à RAM
e) permite uma proteção individual das diferentes zonas do EE do processo (código, dados, *stack*, *heap*), mas pior aproveitamento da RAM por causa da fragmentação interna

8) No escalonamento (de processos) **por fatia de tempo** (*timeslice*), o escalonador substitui o processo em execução por outro processo:

a) apenas quando termina a fatia de tempo (*timeslice*)
b) quando termina a fatia de tempo (*timeslice*) ou o processo em execução pede um I/O
c) apenas quando o processo em execução pede um I/O
d) assim que termina um evento que estava a bloquear um outro processo qualquer
e) assim que um processo mais prioritário fica pronto (READY) para execução

9) Considere uma arquitetura com bus de endereços virtuais (ou lógicos) de 12 bits, endereços reais (ou físicos) de 14 bits e páginas de $256_{10} == 100_{16} == 0001\ 0000\ 0000_2$ bytes. Admita que um dado endereço lógico D se situa na Página 2, deslocamento 32, e que essa página se encontra mapeada na moldura (*frame*) 3. Assinale a opção correta:

a) O endereço lógico é $800_{10} == 320_{16} == 00\ 0010\ 0010\ 0000_2$ e o físico é $544_{10} == 220_{16} == 0010\ 0010\ 0000_2$
b) O endereço lógico é $288_{10} == 120_{16} == 0001\ 0010\ 0000_2$ e o físico é $800_{10} == 320_{16} == 00\ 0010\ 0010\ 0000_2$
c) O endereço lógico é $232_{10} == 0E8_{16} == 0000\ 1110\ 1000_2$ e o físico é $544_{10} == 220_{16} == 00\ 0010\ 0010\ 0000_2$
d) O endereço lógico é $544_{10} == 220_{16} == 0010\ 0010\ 0000_2$ e o físico é $800_{10} == 320_{16} == 00\ 0011\ 0010\ 0000_2$
e) Nenhuma das outras opções está correta

10) Num sistema com suporte para paginação (mas sem suporte a paginação-a-pedido, matéria que não foi lecionada) a tabela de páginas (*page table*) tem como função indicar:

a) para os processos que partilham memória entre si, em que *frames* 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 estão colocadas as páginas desse processo
d) para um processo, em que páginas da memória estão colocadas as *frames* desse processo
e) para um processo, no estado **`READY`**, em que *frames* estão colocadas as páginas desse processo

11) 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();
...
```

12) Qual o número de **novas** threads que estão **activas** no ponto de observação **`<Ponto O>`** quando um processo executa a seguinte sequência de instruções

a) 0
b) 1
c) 2
d) 3
e) 4

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

13) Considere as threads T1 e T2 concorrentemente executadas e duas variáveis globais **`c`** e **`s`**, inteiras e partilhadas, ambas **inicializadas com o valor 0**. Qual é o resultado final que se obtém em **`c`** após execução das instruções inscritas nas "caixas", considerando as seguintes implementações para as duas funções – tranca e destranca – abaixo indicadas, cujo objetivo é garantir uma região crítica (RC) para as atualizações de **`c`**

```
void tranca(int *m) {
  while (*m == 1) ;
  *m= 1;
}

void destranca(int *m) {
  *m= 0;
}
```

T1
```
...
tranca(&s);
  c--;
destranca(&s);
```

T2
```
...
tranca(&s);
  c++;
destranca(&s);
```

a) a alteração de **`c`** pode não acontecer porque as *threads* bloqueiam-se mutuamente (*deadlock*) e isto acontece porque **`s`** devia ser inicializado a 1 (e não a 0)
b) o valor de **`c`** é 0
c) o código está errado, as funções atuam sobre um apontador **`m`** que não existe no programa
d) o valor de **`c`** pode ser 0 ou 1, variando de execução para execução, porque uma RC faz-se destrancando primeiro e trancando depois
e) o valor de **`c`** pode ser -1, 0 ou 1, variando de execução para execução, porque a implementação das funções **`tranca`** e **`destranca`** não garantem uma RC

14) 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

só dava 0 se todos os escritores estivessem fechados

15) Considere o programa abaixo indicado (ignore a falta de *include*s). O valor final da variável **`recur`** (que indica o número de vezes que a função **`fact()`** foi chamada, recursivamente)

```
int recur= 0;

int fact(int x) {
  recur++;
  if ( x > 0 ) return (x * fact(x-1)); else return 1;
}

int main (int argc, char *argv[]) {
  int f= fact(4);
  printf("%d!= %d, %d chamadas recursivas\n", 4, f, recur );
}
```

a) está errado porque o cálculo de **`recur`** está errado
b) está errado porque a execução recursiva não permite a alteração de variáveis globais
c) está errado porque a execução recursiva equivale à execução concorrente de múltiplas *threads*, e como tal seria necessário usar um *mutex* para enquadrar a variável **`recur`** numa região crítica
d) está correto
e) o programa nem sequer passa a compilação

16) Considere o programa abaixo indicado (ignore a falta de *include*s). O que aparece no ecrã é [NOTA: o símbolo "**?**" indica que tanto pode aparecer um 1 como um 2]

```
int v= 0;

void proca(int *x) {
  int i= 0, p;
  p= fork();
  if (p > 0) p=1;
  i++; (*x)++;
  printf("p= %d, i= %d, v= %d\n", p, i, v);
}

int main (int argc, char *argv[]) {
  proca(&v);

  return 0;
}
```

a) não aparece nada porque falta o **`wait()`**
b) o programa dá erro porque só no **`main()`** se pode chamar um **`fork()`**
c) são duas linhas (por qualquer ordem): "`p= 0, i= 1, v= 1`" e "`p= 1, i= 1, v= 1`"
d) são duas linhas (por qualquer ordem): "`p= 0, i= 1, v= ?`" e "`p= 1, i= 1, v= ?`"
e) são duas linhas (por qualquer ordem): "`p= 0, i= ?, v= ?`" e "`p= 1, i= ?, v= ?`"

17) 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) nenhuma das anteriores

18) Considere o programa abaixo indicado (ignore a falta de *include*s). O que aparece no ecrã é [NOTA: o símbolo "**?**" indica que em algumas execuções pode aparecer um 0 e noutras um 1; o símbolo "**??**" indica que em algumas execuções pode aparecer um 1 e noutras um 2]

```
int v= 0;

void *threada(void *x) {
  long i= 0, p= (long)x;
  p++; i++; v++;
  printf("p= %ld, i= %ld, v= %d\n", p, i, v);
  return NULL;
}

int main (int argc, char *argv[]) {
  pthread_t id1, id2;
  pthread_create(&id1, NULL, threada, (void *)0);
  pthread_create(&id2, NULL, threada, (void *)1);
  pthread_join(id1, NULL);  pthread_join(id2, NULL);
  return 0;
}
```

a) não aparece nada porque o programa dá erro de compilação
b) o programa dá erro ao executar quando as duas *threads* escrevem simultaneamente no ecrã
c) são duas linhas (por qualquer ordem): "`p= ?, i= 1, v= ??`" e "`p= ?, i= 1, v= ??`"
d) são duas linhas (por qualquer ordem): "`p= 1, i= 1, v= ??`" e "`p= 2, i= 1, v= ??`"
e) são duas linhas (por qualquer ordem): "`p= 1, i= ?, v= ??`" e "`p= 2, i= ?, v= ??`"

19) O objetivo fundamental do escalonador do CPU num SO moderno (p.ex., Linux ou Windows) que suporta um ambiente gráfico, com janelas, num computador pessoal (*laptop* pu PC) que está ligado a uma rede de comunicações ("net") e é utilizado exclusivamente por um único utilizador é

a) manter a taxa de utilização do CPU o mais alta possível, de forma a minimizar o tempo de execução dos múltiplos processos que estão a correr na máquina, de forma a que estes terminem o mais rapidamente possível
b) privilegiar a execução de processos que realizam tarefas de "segundo plano" (*background*) tais como impressão, transferência (p.ex., *download*) de ficheiros da "net", compilação de programas, etc.
c) privilegiar a execução de processos que realizam tarefas de "primeiro plano" (*foreground*) tais como uso de editores/processadores de texto (Word, notepad) ou outras aplicações interactivas
d) todas as anteriores
e) nenhuma das anteriores

20) O que caracteriza um escalonador MLFQ (*Multi-Level Feedback Queue*) é

a) não usar fatias de tempo
b) favorecer a execução dos processos CPU-bound
c) distribuir equitativamente o tempo de CPU pelos processos
d) escalonar os processos preservando a sua antiguidade (exibindo um carácter FIFO)
e) nenhuma das anteriores

Fundamentos de Sistemas de Operação
1º Teste, 7 de Novembro de 2020

QUESTÕES DE DESENVOLVIMENTO — VERSÃO A

D1) Pretende-se implementar a função exec3 que executa em sequência (um após o outro) três programas recebidos como argumento. Cada programa só pode executar caso a execução de todos os programas anteriores tenha tido sucesso, ou seja, foi possível executá-los e a subsequente execução não terminou com um estado de erro. Por questão de simplificação, os programas a executar não têm argumentos.

Por exemplo, a chamada exec3("ls", "ps", df") lista a diretoria corrente, lista os processos em execução na shell "pai" e mostra o espaço livre em disco, enquanto que a
chamada exec3("ls", "fake", df") apenas lista a diretoria corrente, pois o comando fake não existe.

a) Comece por implementar a função exec1 que permite a um processo executar um programa sem esperar pela sua conclusão. A função deve retornar -1 se houve erro na criação do processo; 0 em caso contrário.

```
int exec1(char* prog) {
    int pid = fork();
     if (pid < 0) {
         return -1;
     } else if (pid == 0) {
         execlp(prog, prog, NULL);
         exit(1);
     }
     return 0;
}
```

b) Implemente agora a função **exec3** que utiliza a função **exec1** para executar cada um dos programas recebidos, garantido que cada um destes só executa depois da confirmação que o anterior executou corretamente.

```
void exec3(char* prog1, char* prog2, char* prog3) {
    int pid = exec1(prog1);
    int status;
    wait(&status);
    if (status == 0) {
      pid = exec1(prog2);
      wait(&status);
      if (status == 0) {
          pid = exec1(prog3);
          wait(NULL);
      }
   }
}
```

D2) Considere um programa para contar o número de pessoas infetadas com Covid-19 num determinado concelho, e calcula a taxa de incidência (número semanal de novos casos por 100000 habitantes) nesse concelho, de modo a avaliar se esses habitantes devem fazer quarentena ou não. Dada a necessidade de rapidez no processamento dos dados, pretende-se implementar uma solução concorrente com base em *threads* POSIX, variáveis de exclusão mútua e primitiva de sincronização *barrier*, para processar essa informação. ***Não pode usar semáforos.***

Para privacidade dos dados, a informação sobre os habitantes lida pelo programa corresponde a um vector de nome *population*, cujas posições contêm um caracter com um de dois símbolos:

i) **'-'** indica que o habitante tem um resultado negativo ao teste COVID-19

ii) **'+'** indica que o habitante tem um resultado positivo

Caso o número total de casos positivos implique uma taxa de incidência perigosa, todas as posições do vector *population* devem ser preenchidas com o caracter **'Q'**. Este vector será posteriormente avaliado para envio de SMS indicando a necessidade de toda a população do concelho entrar em quarentena.

O esqueleto do código do programa descrito em baixo contém alguns espaços em branco que devem ser completados por si. Por simplificação, o programa principal é omitido, assumindo-se que o vector *population* se encontra bem preenchido. É também omitido o código da função *dangerousRate()* que calcula a taxa de incidência e a compara com um valor de risco pré-definido. Assuma também que as variáveis globais já se encontram inicializadas, bem como que não existem erros.

```
// includes...
char *population;       // endereço do vector com o resultado das análises efectuadas
                        // à população de um concelho

int total_population    // comprimento desse vector

pthread_t *ids;         // endereço do vector com os identificadores dos
                        // threads/workers criados

int length_per_thread; // n° de elementos do vector a serem processados por cada
                        // thread

pthread_mutex_t ex = PTHREAD_MUTEX_INITIALIZER;

pthread_barrier_t fillBarrier;

int total_infected= 0; // Contém o número total de infetados num concelho


// Protótipo de uma função a completar, e que está na página seguinte...

void * evaluate_quarantine(void *id);


// Lança o n° de threads passado como argumento, para repartir entre elas a tarefa
// de processar o vector

void evaluate_quarantine_parallel(int nthreads){
     int i;
     // Inicialização da barreira
     pthread_barrier_init(&fillBarrier, NULL, nthreads );

     // Criar os threads
     for(i=0; i < nthreads; i++){
          pthread_create(&ids[i] ,NULL, evaluate_quarantine, (void *)(long)i );
     }

     for(i=0; i < nthreads; i++){
          pthread_join(ids[i], NULL);
     }
}
```

```
// Protótipo da função que calcula a taxa de incidência e devolve se é maior que um
valor de risco pré-definido (é só usar, não se conhece o código)
int dangerousRate(int total_population, int infected);

int is_positive(char result){
 return result == '+';
}

// Função executada por cada thread
void * evaluate_quarantine(void *id){
      int i, local_positives = 0;
      int start = length_per_thread*(int)(long)id;

            int end = start + length_per_thread;
      // processa vector parcial
      for(i=start; i < end; i++){
            if( is_positive(population[i]) ){
                local_positives += 1;
            }
      }
      // actualiza o número total de casos positivos
      pthread_mutex_lock(&ex);
      total_infected += local_positives;
      pthread_mutex_unlock(&ex);

            // Sincronização dos threads
      pthread_barrier_wait(&fillBarrier);
      // thread de identificador zero calcula a taxa de incidência
      if( (int)(long) (int)(long)id == 0 ) {
            if( dangerousRate(total_population, total_infected) )
            quarantine = TRUE;
      }

      pthread_barrier_wait(&fillBarrier);
      // os threads processam o seu vector parcial escrevendo o caracter 'Q'
      // em todas as posições em caso de quarentena do concelho
      if( quarantine ) {
            for(i=start; i < end; i++) population[i] = 'Q';    // send SMS
      }

      return NULL;
}
```

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

```
int fork()
int wait(int *status);


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)


close(int fd)
pipe(int pipefd[2])
```