Fundamentos de Sistemas de Operação
Exame de Época Especial, __ de Setembro de 2020

DURAÇÃO: 2h45 (inclui tolerância)

NOME DO ESTUDANTE: _____ Nº: _____

A duração do exame é 2h45 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 EXAME: (copiar do enunciado)

A) 2 **B) 1** **C) 3** **D) 3** **E) 3** **F) 3** **G) 2**

H) 2

Fundamentos de Sistemas de Operação
Exame de Época Especial, __ de Setembro de 2020

QUESTÕES DE ESCOLHA MÚLTIPLA — VERSÃO A

A) O que, fundamentalmente, distingue um processo de uma *thread* é:

1) Um processo tem as seguintes zonas: código, dados, stack e heap; as threads não têm stack
2) Os processos, por omissão, não partilham os seus espaços de endereçamento entre si; as threads, partilham
3) Um processo pode lançar threads, mas uma thread não pode lançar processos
4) As threads dispõem de mecanismos para exclusão mútua; os processos, não

B) A informação de que se tem um SO de 32-bits instalado no computador significa que:

1) o espaço de endereçamento de um processo tem, no máximo, uma dimensão de 2^{32} = 4GB
2) o SO só gere 2^{32} = 4GB de RAM (mesmo que se tenha mais RAM instalada)
3) cada zona (código, dados, stack, heap) do espaço de endereçamento de um processo tem, no máximo, uma dimensão de 2^{32} = 4GB
4) o espaço de endereçamento de um processo tem, no máximo, uma dimensão de 2^{31} = 2GB

C) Os processadores dispõem de algumas instruções (máquina) ditas *privilegiadas*; estas instruções existem para restringir o acesso a determinados registos, endereços e operações, que só podem ser executadas:

1) por programas escritos directamente em linguagem *assembly*
2) por processos que estão a correr em modo **`root`**
3) quando o processador está no estado **`supervisor`**
4) pelos periféricos, quando estes fazem *interrupts*

D) Considere o código na caixa ao lado; o programa compila correctamente e cria o executável, e o conteúdo do ficheiro **`abc.txt`** é a sequência de caracteres **`abc123`**. O que aparece no ecrã quando se executa?

```
int main() {
  char buf1[4], buf2[4]; int fd;

  fd= open("abc.txt", O_RDONLY);

  if (fork()) {
    read(fd, buf1, 3); buf1[3]='\0';
    printf("Buffer 1: %s\n", buf1);
  } else {
    read(fd, buf2, 3); buf2[3]='\0';
    printf("Buffer 2: %s\n", buf2);
  }
...
}
```

1) Uma mensagem de erro devida ao **`read()`** executado pelo processo filho, porque o ficheiro só está aberto para o pai. O print feito pelo pai aparece e vê-se a *string* **`abc`**.
2) Aparece primeiro a *string* **`abc`** e depois a **`123`**.
3) Aparecem as *strings* **`abc`** e **`123`** mas por uma ordem qualquer (quando se executa várias vezes o programa)
4) Aparecem "misturas" entre os caracteres **`abc`** e **`123`**, em número e ordem qualquer (podem aparecer só números, só letras, ou misturas dos dois) que podem variar quando se executa várias vezes o programa.

E) Numa descrição resumida, a implementação da chamada **`fork()`** consiste no seguinte:

1) É criada uma nova tabela de páginas (PT) para o processo filho e depois, no primeiro acesso a cada página, o conteúdo da página do processo-pai é copiado integralmente para a *frame* alocada para o filho, sendo o endereço da *frame* inserido na PT.
2) É criada uma nova tabela de páginas (PT) para o processo filho e depois, no primeiro acesso a cada página, o conteúdo da página é copiado da correspondente zona do programa executável em disco para a *frame* alocada para o filho, sendo o endereço da *frame* inserido na PT.
3) É criada uma nova tabela de páginas (PT) para o processo filho, tabela essa que é uma cópia da PT do pai e depois, no primeiro acesso em escrita a uma dada página, uma nova *frame* é alocada para o filho, o endereço dessa *frame* é inserido na PT, e é efectuada a cópia da correspondente página do pai para essa *frame*.
4) É criada uma nova tabela de páginas (PT) para o processo filho, tabela essa que é uma cópia da PT do pai apenas para as páginas *read-only* (de código e constantes); para as páginas *read-write*, uma nova *frame* é alocada para cada página do filho, o endereço dessa *frame* é inserido na PT, e é efectuada a cópia da correspondente página do pai para essa *frame*.

F) Num computador moderno (por exemplo, um laptop), os tempos de acesso a: registos do processador (abrev. P), posições de memória não-cached (abrev. M), posições em cache de nível 1 (abrev. C), blocos de disco HDD *(i.e.*, disco magnético, e blocos não-cached) (abrev. H), e blocos de disco SSD *(i.e.*, disco de estado-sólido, e blocos não-cached) (abrev. S), são: [**NOTA**: indicam-se aqui **ordens de grandeza**, e <u>não valores exactos</u>! As abrev. usadas são as usuais: n-nano; m-mili; u-micro.. O sinal ~ é usado para reforçar o carácter aproximado]

1) P < 1 us; M ~ 50 ns; C ~ 1 us; H ~ 10 ms; S ~ 0,1 ms
2) P < 1 ns; M ~ 50 ns; C ~ 1 us; H ~ 10 ms; S ~ 0,1 ms
3) P < 1 ns; M ~ 50 ns; C ~ 1 ns; H ~ 10 ms; S ~ 0,1 ms
4) P < 1 ns; M ~ 50 ns; C ~ 1 ns; H ~ 10 ns; S ~ 0,1 ns

G) A largura de banda (débito, ou taxa de transferência) de um volume RAID-5 formado por 5 discos idênticos, cada um com um débito individual D e acedido sequencialmente <u>em leitura</u> é aproximadamente,

1) 5 x D
2) 4 x D
3) 3 x D
4) D

H) Na programação de *drivers* para periféricos, podem usar-se duas técnicas distintas, sendo uma delas designada espera activa. O que essencialmente caracteriza a programação por espera activa é:

1) O periférico está continuamente a executar o programa (*driver*) para ler repetidamente o registo de estado até que uma dada condição (bit) indique a conclusão da operação em curso. Enquanto tal não se verifica, o programa (*driver*) está parado e o processador pode ser usado para outras tarefas; assim que se verificar, a execução do *driver* é retomada, o fim da operação é tratado, e o *driver* termina.

2) O processador executa o programa (*driver*) para ler repetidamente o registo de estado do periférico até que uma dada condição (bit) indique a conclusão da operação em curso. Quando tal acontece, o fim da operação é tratado e o *driver* termina. Só então o processador pode ser usado para outras tarefas.

3) O periférico executa o programa (*driver*) para ler repetidamente o registo de estado até que uma dada condição (bit) indique a conclusão da operação em curso. Enquanto tal não se verifica, o programa (*driver*) está parado e o processador está a ser usado para outras tarefas. Assim que se verificar a conclusão da operação, é desencadeada uma interrupção, a execução do *driver* é retomada, o fim da operação é tratado, e o *driver* termina.

4) O processador executa o programa (*driver*) para ler o registo de estado até que uma dada condição (bit) indique a conclusão da operação em curso. Enquanto tal não se verifica, o programa (*driver*) está parado; assim que se verificar, é desencadeada uma interrupção e a execução do *driver* continua, o fim da operação é tratado e o *driver* termina. Contudo, enquanto o programa *driver* está parado, o processador pode ser usado para outras tarefas.

Fundamentos de Sistemas de Operação
Exame de Época Especial, __ de Setembro de 2020

QUESTÕES DE DESENVOLVIMENTO

D1) Considere um programa para achar a soma dos elementos existentes num vetor, e que utiliza programação paralela com base em `pthreads` para dividir o trabalho. O esqueleto do código do programa abaixo exibido contém alguns espaços em branco que devem ser completados por si. Por simplificação, o programa principal é omitido, pelo que apenas tem de completar a função que lança o conjunto de *threads* bem como a função que é executada por cada thread. Assuma também que as variáveis globais já se encontram inicializadas, bem como que não existem erros.

NOTA de correcção: aceitar acessos como vector e casts imperfeitos ou inexistentes

```
// includes...
#define SIZE 1000
int *array, length;          // endereço do vector a processar e comprimento do vector

pthread_t *ids;              // endereço do vector com os identificadores dos threads criados
int length_per_thread;   // número de elementos do vector a serem processados por cada thread
pthread_mutex_t ex = PTHREAD_MUTEX_INITIALIZER;
int soma_final = 0;       //  Contém a soma total (final) dos elementos do vector

// Função executada por cada thread
void * soma_parcial(void *id){
   int i, local_sum = 0;  // utilizado por cada thread para a soma parcial dos elementos que trata
   int start = length_per_thread*(int)(long)id;          // elemento do vector onde a thread começa a somar
   int end = [ start + length_per_thread - 1 ]           // idem, onde a thread termina a soma parcial      10%

   // processa vetor parcial
   for (i=start; i < end; i++) {
         local_sum = [ local_sum + array[i] ]                                                              10%
   }

   // atualiza a soma total
         [ pthread_mutex_lock(&ex) ]                                                                       15%
          soma_final = [ soma_final + local_sum ]                                                          10%
         [ pthread_mutex_unlock(&ex) ]                                                                     15%
   return NULL;
}

// Lança o número de threads passado no argumento nthreads para processar o vetor, imprime o valor final
void soma_elementos_em_paralelo (int nthreads) {
   int i;
   for(i=0; i < nthreads; i++)
         pthread_create( [ ids[i] ] , NULL, soma_parcial, [ (casts) i ] );                                 20%

   // esperar que todas as threads terminem antes de imprimir o valor
      [ for(i=0; i < nthreads; i++) pthread_join( ids[i], NULL) ]                                          20%
   printf("Soma dos elementos do vector= %d\n", soma_final);
}
```

D2) Considere um computador no qual se executa um sistema operativo de "tipo UNIX" que suporta **memória virtual através de paginação-a-pedido**. O processador tem um **bus de endereços lógicos de 32 bits**, uma unidade de tradução de endereços (MMU) equipada com um TLB (*Translation Lookaside Buffer*) que traduz os endereços lógicos em físicos segundo o conteúdo de uma tabela de páginas (*page table*). O **bus de endereços físicos** (que vai da MMU/TLB para a RAM) **tem 24 bits**. Assuma que o SO usa uma página de 512 bytes.

a) O que é o Espaço de Endereçamento (EE) de um processo?

É o conjunto de posições que memória (lógica ou virtual) que podem ser acedidos pelo processo - podendo considerar-se que as posições inválidas no interior de um EE linear fazem parte do EE

b) Qual é a dimensão máxima do EE de um processo na arquitectura acima descrita? Justifique.

2^32 = 4 GB, que é a dimensão do bus lógico

c) Qual é o número máximo de frames suportado pela arquitectura acima descrita? Justifique.

2^24 = 16 MB, que é a dimensão do bus físico / 2^9 = 512 que é a dimensão da frame
2^24 / 2^9 = 2^15 = 32768 frames

d) O que é a tabela de páginas de um processo? Ou seja, para que serve uma tabela de páginas, e qual é o seu conteúdo (i.e., que informação contém)?

A tabela de páginas (PT) de um processo é uma estrutura de dados de tipo tabela, mantida pelo SO e que, para cada processo indica: em que frame da memória está cada página desse processo (ou tem um valor que indica que não está); para essa página, quais são as permissões de acesso (leitura, escrita, execução, inválida = sem acesso)

D3) Complete os espaços em branco de um programa que se pretende que lance dois processos; o que lança é naturalmente o pai, e o outro, o filho. Um dos processos escreve num pipe a frase "ola", e o outro lê os caracteres e coloca-os num buffer cujo conteúdo é depois impresso no ecrã. Há dois espaços em branco que devem ser usados para garantir (sem declarar novas variáveis!) que o programa funciona sempre correctamente em qualquer situação (i.e., o que lê arranca, lê e termina antes que o que escreve ter oportunidade de escrever, ou vice-versa, ou …). Não é preciso tratar situações de erro.

```
int main(int argc, char *argv[]) {
  int fd[ 2 ];                          // estrutura para os fd's do pipe   20%
  int pid; char buf[4];
  pipe(fd)                              ;     // cria pipe                  20%

  pid=fork();
  if (pid) {

        // escrever "pai" ou "filho" no espaço e branco para indicar este caso
    printf("Sou o processo pai , PID:%d\n", getpid() );                     10%

    close(fd[0])                                    // para que ... escrito seja
    10%                                             // depois correctamente lido

    write(fd[ 1 ], "ola", 3);     // processo escreve 3 caracteres no pipe  15%

  } else {
        // escrever "pai" ou "filho" no espaço e branco para indicar este caso
      printf("Sou o processo filho , PID:%d\n", getpid() );                 10%

      close(fd[1])                          // para que a leitura funcione
                                            // sempre correctamente

      read(fd[ 0 ], buf, 3);      // este processo lê 3 caracteres do pipe  15%

      buf[3]= '\0';
      printf("Tenho buf=%s\n",buf);
  }

  return 0;
}
```

D4) Considere um sistema de ficheiros (SF) como o **`ext3`** ou outro similar da "família" dos SF nativos do Linux (NOTA: para efeitos desta questão ignore tudo o que se refere a *journaling*). O disco físico sobre o qual o SF está implementado tem blocos de 512 bytes.

a) Como é que o SF faz a gestão do espaço (dos blocos de disco) livre/ocupado? Refira que estrutura(s) de dados é(são) usada(s), e como são actualizadas à medida que "objectos" do SF tais como ficheiros e directorias são criados, apagados, etc.

Cada SF do tipo ext tem uma estrutura de dados designada bitmap de blocos que indica se um bloco está livre ou ocupado. [Podemos imaginar que] todos os blocos [que nos interessa considerar para efeitos desta questão] estão, inicialmente, marcados como livres. [Não é necessário especializar os bitmaps em BM de inodes e de data blocks]

Cada vez que um "objecto" do SF - ficheiro, directoria, inode, etc. - necessita de um bloco para armazenar informação, o bitmap é consultado e um bit livre é marcado como ocupado e o correspondente bloco é utilizado. Quando o inverso - destruição de um objecto, ou outra operação que liberta blocos - acontece, os bits correspondentes aos blocos libertados são marcados como livres.

b) Em que consiste a operação usualmente designada por *formatação*, num SF neste tipo?

Consiste em inicializar (com os valores apropriados) as estruturas de dados que são usadas para gerir o SF: os bitmaps são inicializados indicando que os blocos estão todos [excepto aqueles que contêm as próprias estruturas] livres, os inodes estão todos livres [excepto os usados para a directoria raiz do SF e os ficheiros especiais ., .., etc.]

Essa operação pode, eventualmente, testar os/alguns blocos do disco.

c) Como é, neste tipo de SF, implementada a estrutura de dados que vulgarmente designamos por directoria?

Uma directoria é um ficheiro; só é especial na medida em que o seu conteúdo só pode ser lido/escrito pelo SO/SF, de forma que um utilizador é obrigado a usar uma API especial para directorias [readdir(), creat(), …]. Logicamente o ficheiro é constituído por entradas que são pares (nome-do-objecto, inode), onde "nome-do-objecto" se refere ao nome de um ficheiro ou directoria existente "dentro" da directoria em questão, e o inode indica a restante informação sobre esse objecto.

D5) Descreva sucintamente como é que, num SO "de tipo UNIX" (Linux ou similar, mas uma versão muito simples, ou primitiva – ver simplificações abaixo), o SO decide qual o processo que vai ser executado. Assuma as seguintes simplificações: i) a máquina só tem um processador; e ii) o SO só suporta processos, não suporta *threads*.

Para uma boa resposta, aborde pelo menos estes tópicos: quais os estados de um processo; qual(is) a(s) estrutura(s) de dados fundamentais para preservar informação sobre a execução de um processo; que informação é nelas guardada; como e quando o SO decide executar um processo; se, quando um processo está a ser executado (e portanto ainda não terminou) pode "sair" da execução para dar lugar a outro, etc…

O módulo do SO que decide qual o próximo processo a ser executado é o escalonador (scheduler); este implementa uma política de escalonamento, e essa política mantém os processos (representados pelos PCBs - Process Control Block) em filas de espera.

Se o computador só tem um CPU então só pode haver, em cada instante, um processo no estado RUNNING; os outros, ou estão READY (podem correr, estão à espera do CPU) ou WAITING (estão à espera de um recurso/evento - só poderão passar para READY quando tal acontecer).

Na fila READY [podemos imaginar] os processos estão ordenados por ordem da sua importância: quem está "à frente" terá direito ao CPU antes de quem "está atrás". O que determina a posição da fila depende da política usada[: pode ser a antiguidade (FIFO), a prioridade, etc…]

Os SOs modernos usam timeslices: um processo corre até acabar o TS, ou até fazer uma operação que o bloqueia. Nestes casos, o escalonador é reactivado e decide se o processo que estava a correr pode continuar (pois é ainda o mais importante) ou se vai sair para dar vez a outro (PREEMPTED).

[Não é necessário descrever o process switching]

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, valorInicial)
sem_post(sem_t *var)
sem_wait(sem_t *var)


close(int fd)
pipe(int pipefd[2])
read(int fd, void *buffer, size_t nbytes)
write(int fd, const void *buffer, size_t nbytes)
```