MIEEC **Sistemas Operativos – 2015/16** **FEUP**

Tipo de Prova: consulta de "*cheat-sheet*" **Exame da Época Recurso**
Duração: 2 horas **11.Julho.2016**
Cotação máxima: 20 valores
Estrutura da prova: Parte I (escolha múltipla, 30%); Parte II (mais convencional, 50%); Parte III (predominantemente de aplicação, 20%)

PARTE I: Escolha múltipla [6 val.]

Utilização: para cada pergunta só há uma resposta correcta; indique-a (com a letra correspondente) na folha de respostas, completando uma tabela semelhante à que se segue; se não souber a resposta correcta, nada preencha ou faça um traço nessa alínea.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Cotação: cada resposta certa vale 1 ponto; cada resposta errada vale – 0,5 ponto (note o sinal menos!); cada resposta ambígua, ininteligível ou não assinalada vale 0 ponto. O total é 15 pontos (que irão equivaler aos valores indicados para esta parte da prova).

1. Um sistema operativo é um "simulador de máquina virtual". Isso quer dizer que:

A) facilita a programação de aplicações.

B) facilita a utilização de aplicações.

C) facilita o teste de aplicações.

2. As variáveis de ambiente costumam ser utilizadas para fornecer

A) informação comum a um conjunto de programas.

B) informação específica a cada instância de um programa.

C) informação opcional a uma classe específica de programas.

3. Para se compilar um *character device driver* usa-se o utilitário `make` com uma *makefile* apropriada. Assim é mais prático porque

A) a mesma *makefile* dá para todos os *drivers*.

B) pode usar-se o *kernel build system*.

C) evita-se a repetição da linha de compilação: "`cc -Wall -c device.c -o device.ko`".

4. Concorrência de processos quer dizer

A) execução "simultânea" de vários processos, provavelmente competindo pelos mesmos recursos.

B) execução simultânea de vários processos em vários processadores.

C) execução "simultânea" de vários processos de um só utilizador.

5. Um processo tem associado:

A) um programa, uma zona de memória e um ficheiro aberto.

B) um programa, uma zona de memória e uma entrada na tabela de processos.

C) um programa, um processador e um dono.

6. Na sequência da invocação da chamada `fork()` o novo processo irá começar imediatamente a executar.

A) Não, pois o processo-pai é sempre o primeiro a executar.

B) Sim, se for escolhida a opção certa para **`fork()`**.

C) Não é possível afirmar isso com certeza.

7. Na programação com *threads* POSIX, tem de se associar a cada *thread* uma rotina.

A) Sim e essa rotina poderá, ou não, ser diferente para cada *thread*!

B) Sim e essa rotina terá necessariamente de ser diferente para cada *thread*!

C) Tal não é necessário, pois cada novo *thread*, por omissão, fica associado a **`main()`**.

8. Uma região crítica consiste num segmento de código que

A) pode ser executado por vários processos em simultâneo.

B) só pode ser executado em simultâneo por vários *threads*.

C) não pode ser executado em simultâneo.

9. Uma das estratégias para se lidar com encravamentos ao nível do sistema operativo preconiza

A) detectar-se o problema e tentar resolvê-lo, eventualmente terminando alguns processos.

B) evitar-se o problema, planeando sempre tudo com os utilizadores.

C) prevenir-se o problema, negando o pedido de memória a alguns dos processos.

10. Num sistema de memória virtual paginada, cabe à Unidade de Gestão de Memória (MMU) do processador a indicação

A) do tamanho de página apropriado à situação.

B) da moldura a esvaziar, no caso de se dar uma falha de página.

C) do endereço físico de uma referência à memória.

11. Um sistema de memória virtual:

A) tem de utilizar paginação.

B) pode utilizar paginação ou segmentação.

C) não pode utilizar paginação sem segmentação.

12. Registos, *cache* (de processador), RAM principal, disco magnético, são tipos de "memória" cuja ordem de citação corresponde, tipicamente, a uma ordem

A) crescente de rapidez.

B) crescente de capacidade.

C) crescente de preço.

13. A formatação de baixo-nível de discos rígidos basicamente consiste

A) em se efectuar a laser infravermelho uma série de traços na camada superior do disco magnético.

B) em se escrever determinada informação magnética nas partes periféricas e centrais do disco.

C) em se efectuar a escrita do preâmbulo (ou cabeçalho) de cada sector do disco magnético.

14. Os ficheiros costumam ser guardados nos discos em blocos cujo tamanho

A) é ajustado à rapidez dos discos.

B) é principalmente dependente do tamanho dos sectores dos discos.

C) é um compromisso entre eficiência de armazenamento e taxa de transferência.

15. Quando um processo pede para abrir um ficheiro, o sistema de ficheiros começa por

A) consultar a entrada do directório onde o ficheiro está armazenado.

B) consultar a entrada do directório onde o ficheiro está listado.

C) encontrar o primeiro bloco do ficheiro; só depois consulta a entrada do directório o ficheiro está listado.

PARTE II: Mais convencional [10 val.]

Utilização: cada resposta, a apresentar na folha de respostas, deve ser (brevemente) justificada.
Cotação: mostrada em cada pergunta.

1. **[1 val.]** No sequência da avaliação do *Serial Port Device Driver* que desenvolveu, explique como efectuaria o teste de transmissão de um ficheiro enviado do sistema na máquina hospedeira para o sistema com o *Device Driver* instalado (máquina virtual). Poupe as palavras: é suficiente a indicação clara e concreta da sequência de comandos a utilizar em terminais de texto em ambos os lados.

2. **[1 val.]** As funções seguintes são normalmente usadas na função de saída de um módulo de núcleo de Linux. (A função de saída é invocada quando o módulo é removido.)

```
cdev_del(ptr_cdev);
free_irq(SERIAL_INT, dev_id);
outb(0x00, SERIAL_BASEPORT+UART_IER);
release_region(SERIAL_BASEPORT, NUM_SERIAL_PORTS);
unregister_chrdev_region (dev, num_minor);
```

   a) Escolha as que usaria no módulo em *polled operation* e apresente-as pela ordem em que devem ser invocadas;

   b) explique o significado dos parâmetros de invocação das funções que escolheu.

3. **[1 val.]** Citando o livro *Linux Device Drivers*, Corbet et al., ed. 3, p. 117:
   «*Spinlocks are, by their nature, intended for use in multiprocessor systems...*»
   Esclareça a afirmação, eventualmente, usando o método de "redução ao absurdo".

4. **[1 val.]** O seguinte extracto de código é executado concorrentemente por `N` *threads*, cada um incrementando a variável global `count` `MAX` vezes e usando a variável partilhada `busy` para promover a sincronização. Estas variáveis estão definidas e inicializadas conforme mostrado.
   Comente a previsão do programador em conseguir que `count` atinja o valor `N*MAX`.

```
1. long count = 0; // global
2. int busy = 0; // global
3. ...
4.
```

```
5. for (i = 0; i < MAX; i++) {
6.   while(busy)
7.           ; // espera activa
8.      busy = 1;
9.   count++;
10.  busy = 0;
11.    }
12.  ...
```

5. **[1 val.]** Pode alguma das seguintes operações ser executada em "modo de utilizador" (*user mode*)?

   a) Diminuição da prioridade de execução de um processo.

   b) Remoção de inibição de interrupções.

6. **[1 val.]** O seguinte segmento de código foi escrito com a intenção de fazer aparecer no ecrã a mensagem "`Hello, world!`".

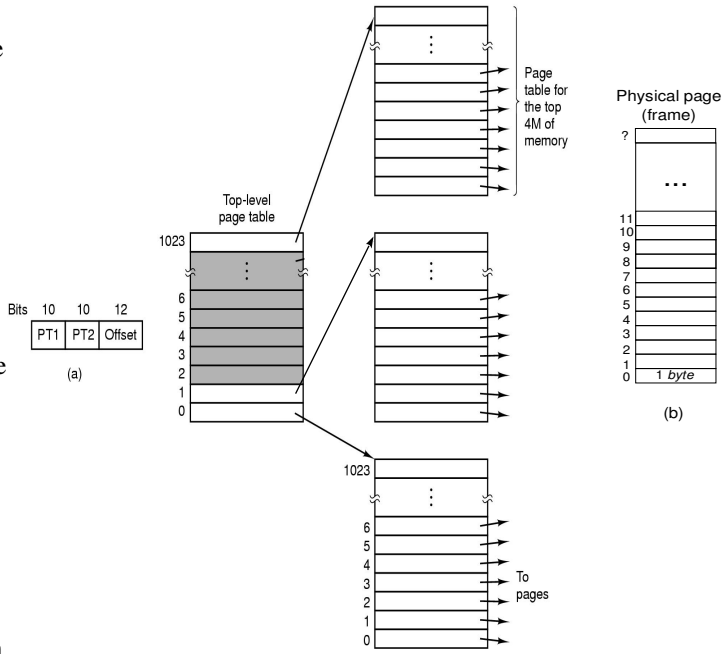
```
1.   ...
2.   void *fthr(void *a) {
3.    printf("Hello, ");
4.    return (NULL);
5.   }
6.
```

```
7.  int main() {
8.     pthread_t pt;
9.   pthread_create(&pt, NULL, fthr, NULL);
10.  printf("world!");
11.  return(1);
12.  }
```

   a) Será o resultado, em geral, o esperado?

   b) Refaça o programa usando processos em vez de *threads*, mantendo a ideia de divisão e colaboração na tarefa de impressão e garantindo a escrita da referida frase.

7. **[1 val.]** Considere a implementação de memória virtual paginada representada no esquema (a) ao lado, retirado dos acetatos das teóricas, e referente a um dado processo. Sabe-se que cada entrada de uma tabela de páginas usa 4 bytes e que cada tabela de páginas ocupa uma página de memória.
   a) Como classificaria a implementação ilustrada: linear, multi-nível (2 níveis) ou multi-nível (três níveis)?
   b) Qual é o tamanho das páginas do sistema ilustrado?
   c) Qual é a máxima quantidade de memória a que o processo tem acesso?
   d) Supondo que em valores decimais `PT1 = 0`, `PT2 = 5` e `Offset = 9`, complete todo o esquema representado, (a) e (b), com setas apropriadas ilustrando claramente qual é o byte endereçado na página/moldura física mostrada em (b).



8. **[1 val.]** Como é que a implementação de memória virtual paginada reduz significativamente o número médio de acessos à memória necessários para realizar a conversão de endereços?

9. **[1 val.]** Os discos magnéticos modernos mantêm desde há anos a configuração interna no que concerne à geometria do armazenamento físico dos dados. Com auxílio de um esquema, explique o que se entende pelos seguintes termos, a sua inter-relação e a maneira como são mapeados no interior de um disco:
   a) cabeça (*head*) de leitura/escrita;
   b) cilindro;
   c) pista (*track*);
   d) sector.

10. **[1 val.]** Considerando a implementação de directórios em sistemas de tipo Unix, explique:
   a) onde são mantidos os atributos dos ficheiros referenciados num directório;
   b) como é possibilitada a partilha de ficheiros por via de *hard links*.

PARTE III: Predominantemente de aplicação [4 val.]

Utilização: cada resposta, a apresentar na folha de respostas, deve ter um breve comentário justificativo.
Cotação: mostrada em cada pergunta.

1. **[2 val.]** A variável de ambiente `PATH` está disponível a qualquer processo e contém uma lista de directórios (separados por "`:`") que são percorridos pela biblioteca de sistema para encontrar o ficheiro cuja execução é pedida via `execlp()` ou `execvp()`. Normalmente, a lista é percorrida pela ordem natural, mas neste problema quer fazer-se uma pesquisa "em paralelo" dos directórios listados.
   Analise o programa (simplificado) abaixo listado: obtém da linha de comando o nome de um ficheiro executável e separa os componentes (directórios) de `PATH`, que vão surgindo no ponto A. O ciclo `for()` termina quando esgotarem os directórios de `PATH`.
   a) Complete o ciclo no ponto `A` com código de criação de processos "filho" que irão executar a função `indir()` por forma a que, caso *file* esteja listada em *dir*, seja dada ao sistema ordem para executar *file*.
   Nota: não é necessário escrever o código de `indir()`!
   b) Suponha que se constituiu uma zona de memória partilhada por todos os processos gerados pelo programa listado abaixo; nessa zona partilhada pode-se criar e utilizar variáveis inteiras e mutexes de forma semelhante ao que se usa com *threads*.
   Como pode haver vários executáveis com nome *file* em vários dos directórios componentes de `PATH`, escreva código que garanta que, na sequência da pesquisa em paralelo via `indir()`, só um dos executáveis é posto em execução.

```
1.  // find_exe: pesquisa directórios em paralelo para encontrar executável
2.  #include ...
3.
4.  int indir (char *file, char *dir) {
5.  // código que verifica se "file" está listado em "dir"
6.  //   retorna 1 se estiver e 0 se não estiver
7.  } // indir()
8.
9.  int main(int argc, char *argv[], char *env[]) {
10.       char *path;     // apontador para variável de ambiente PATH
11.  if (argc < 2) {
12.       fprintf(stderr, "\nUsage: %s <executable_name>\n", argv[0]);
13.       exit(1);
14.  }
15.  path = getenv("PATH");
16.  // extracção dos dirs do PATH:
17.  char *pi, *token, *dir;
18.  pi = malloc (strlen(path)+1);
19.  pi[0] = '\0';
20.  strcpy (pi, path);  // because strtok changes 1st argument! :-(
21.  for ( ; ; pi=NULL) {
22.       token = strtok(pi, ":");
23.       if (token == NULL)
24.            break;
25.       dir = malloc (1+strlen(token));
26.       dir[0] = '\0';
27.       strcpy (dir, token);
28.
29.       // A: dir aponta para um dos directórios listados em PATH
30.
31.       } // for()
32.  return 0;
33.  } // main()
```

2. **[2 val.]** Pretende-se escrever na saída padrão (*stdout*) a frase "`Hello my friends!`". Para o efeito deve-se usar três *threads* independentes (*detached*), criados explicitamente em `main()`, e de forma a que cada um escreva uma das três palavras. Todos os 3 *threads* executam a mesma função, `thr_func()`, em concorrência e devem coordenar-se usando primitivas de sincronização.
   a) Escreva um programa que cumpra os requisitos apresentados, com liberdade para usar "esperas activas" (*busy waiting*).
   b) Escreva um outro programa que cumpra os requisitos apresentados, mas que não use "esperas activas" (*busy waiting*).

JMMC

Protótipos

Processos

```
pid_t fork(void);
int execve(const char *filename, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
void _exit(int status);
      WIFEXITED(); // macro
      WEXITSTATUS(); // macro
char *getenv(const char *varname);
int setenv(const char *varname, const char *value, int overwrite);
      (The strings in the environment list are of the form name=value.)
```

Threads

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
      void *(*start_routine)(void *), void *arg);
void pthread_exit(void *retval);
int pthread_join(pthread_t th, void **thread_return);
pthread_t pthread_self(void);
int pthread_detach(pthread_t thread);
```

Mutexes/Locks

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex,
const pthread_mutex_attr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Variáveis de Condição

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Semáforos

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
int sem_destroy(sem_t *sem);
```

Sistema de Ficheiros

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
      flags: O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_APPEND
      mode: S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP,
S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fildes, off_t offset, int whence);
      whence: SEEK_SET, SEEK_CUR, SEEK_END
```

Sinais

```
int kill(pid_t pid, int sig);
int pause(void);
int sigsuspend(const sigset_t *mask);
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```