MIEIC **Sistemas Operativos – 2020/21** **FEUP**

Prova com consulta apenas da documentação fornecida **Exame da Época de Recurso**
Duração: 1,5 horas **23.Junho.2021**
Cotação máxima: 50 pontos ; peso na nota final da disciplina: 50%
Estrutura da prova: escolha múltipla

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.

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 50 pontos, que irão equivaler a 20 valores.

Se desejar, *poderá* fazer um pequeno comentário em alguma pergunta que lhe pareça ambígua. No caso de a sua resposta ser considerada errada, o seu comentário *poderá* ajudar a perceber a sua ideia e *poderá* minorar a penalização da classificação.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			16	17	18
														a	b	c			

19	20	21		22	23	24	25		26		27	28	29	30	31	32	33	34	
		a	b				a	b	a	b								a	b

35		36	37	38		39	40	41	42
a	b			a	b				

---

1. Um programa só pode executar num dado sistema se neste estiverem disponíveis bibliotecas dinâmicas.

A) Sim, é verdade para todos os casos de programação em C e C++.

B) Em princípio não, desde que o programa tenha sido compilado com bibliotecas estáticas.

C) Não, não são precisas bibliotecas. Basta ver a linha de compilação do `helloWorld.c`:

```
gcc -Wall -o helloWorld helloWorld.c
```

2. As variáveis de ambiente costumam ser utilizadas para fornecer

A) informação que pode ser usada por um programa e seus descendentes (árvore de processos).

B) informação específica à instância de um programa executado.

C) informação opcional a programas *multithread*.

3. Muitas *makefiles* começam com o objectivo "`all`" e terminam com o objectivo "`clean`".

A) Isso só acontece nas que são usadas para a construção de programas em C e C++.

B) Não é assim que tem de ser: em programação, só o "`all`" é obrigatório.

C) Isso é mais costume do que imposição! Em programação, dá jeito ter objectivos como esses.

4. Academicamente, recomenda-se que a arquitectura de um sistema operativo moderno seja modular, à volta de um *micro-kernel*.

A) Sim, mas por questões de segurança muitos sistemas modernos são, realmente, monolíticos.

B) Sim, mas por questões de eficiência muitos sistemas modernos são, realmente, monolíticos.

C) Sim, mas só se todos os serviços de sistema executarem no modo de supervisão (privilegiado).

5. Um processo executa num computador moderno como que de uma forma contínua, ocupando sempre o processador.

A) Sim, mas só no caso dos processos especiais.

B) Sim, do ponto de vista do processo.

C) Sim, mas apenas quando só há um utilizador.

6. Um sistema multi-programação destina-se a

A) ser programado múltiplas vezes.

B) executar múltiplos programas em simultâneo.

C) ser usado por múltiplos utilizadores em simultâneo.

7. Num sistema operativo, uma "chamada ao sistema"

A) permite a invocação de uma qualquer rotina do sistema operativo.

B) permite à camada de supervisão pedir um serviço à camada de utilização.

C) permite à camada de utilização pedir um serviço à camada de supervisão.

8. Em Unix, quando um processo pretende executar um programa diferente do inicial, terá sempre de invocar a chamada `fork()`.

A) Não, se se pretender descontinuar o programa inicial. Basta, então, invocar-se `execve()` ou uma variante.

B) Sim, essa é uma característica da gestão de processos em Unix. Depois do `fork()`, poderá invocar-se `execve()` ou uma variante.

C) Não é necessário usar `fork()`, basta invocar a chamada ao sistema `execve()`, ou uma variante, e um novo processo será lançado.

9. Considere o excerto de código junto. Quantas vezes será impresso "`Hello World`" na saída padrão?

```
   int i;
void main() {
   for (i=0; i<2; i++)
      fork();
   printf("Hello World\n");
}
```

A) 3

B) 4

C) ... [Imprime até haver exaustão de recursos]

10. Pelo excerto de código junto, o correspondente programa, gerado e executado, vai enviar a outro processo (de identificador recebido na linha de comando) uma série de sinais.
De entre as opções apresentadas, escolha a que reflecte a provisão que o programa do processo alvo tem de ter para não ser terminado abruptamente.

```
void main(int argc, char *argv[]) {
   int pid = atoi (argv[1]);
   kill(pid, SIGUSR1);
   kill(pid, SIGINT);
   kill(pid, SIGCHLD);
}
```

A)
```
void func1(int signo) {  }
signal(SIGUSR1, func1);  signal(SIGINT, func1);  signal(SIGCHLD, SIG_DFL);
```

B)
```
void func1(int signo) {  }  ;  void func2(int signo) {  }
signal(SIGUSR1, func1);  signal(SIGINT, SIG_DFL);  signal(SIGCHLD, func2);
```

C)
```
void func1(int signo) {  }
signal(SIGUSR1, SIG_DFL);  signal(SIGINT, func1);  signal(SIGCHLD, SIG_IGN);
```

11. Ao escrever um programa *multithread* genérico, está ao cuidado do programador:

A) o escalonamento das *threads* (utilização do processador).

B) o controlo do acesso das *threads* à memória partilhada.

C) o estabelecimento de canais de comunicação (*pipes*) entre as *threads*.

12. Tanto se pode usar `sigaction()` como `signal()` para definir a rotina de tratamento de um sinal, mas uma das vantagens do uso de `sigaction` é que:

A) permite a entrega de sinais diferentes a diferentes *threads*.

B) permite o bloqueio de outros sinais aquando da execução da rotina.

C) permite ignorar o sinal `SIGKILL`.

13. Compilou-se e pôs-se em execução o programa de código junto. Cerca de 5 segundos após o arranque, premiu-se `CTRL+C`. Na saída padrão foi impresso:

```
void main() {
   printf("%u", sleep(10));
}
```

A) [nada].

B) 5 [exactamente].

C) 5 [aproximadamente].

14. No escalonamento de processos em sistemas com interfaces gráficas associadas, costuma

A) dar-se mais prioridade aos processos interactivos.

B) dar-se um *quantum* maior aos processos interactivos.

C) dar-se mais registos do processador aos processos interactivos.

15. Num sistema operativo em que o escalonamento do processador é do tipo "à vez" (*round-robin*) correm vários processos, maioritariamente *I/O bound*. Admitindo que *C* = duração média da comutação de contexto e *B* = duração média dos picos de processamento (*CPU-bursts*) dos processos, o que diria do desempenho global do sistema para cada uma das escolhas do valor do *quantum* (*Q*) atribuído igualmente a todos os processos?

a) $Q \approx C$:

A) Bom desempenho geral, para pequenos valores de *C*.

B) O sistema não vai responder rapidamente aos utilizadores.

C) Desempenho vai depender fortemente do valor de *C*.

b) $Q \approx B$:

A) Bom desempenho geral, para pequenos valores de *C*.

B) O sistema não vai responder rapidamente aos utilizadores.

C) Desempenho fraco, mesmo para pequenos valores de *C*.

c) $Q > B$:

A) Bom desempenho geral, para pequenos valores de *C*.

B) O sistema não vai responder rapidamente aos utilizadores.

C) Desempenho fraco, pois penaliza a maioria dos processos (*I/O bound*).

16. Concorrência de processos quer dizer

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

B) execução simultânea de vários processos cada um no seu processador.

C) execução “simultânea” de vários processos de um só utilizador.

17. Uma região crítica consiste num segmento de código que os processos ou *threads* de um programa

A) poderão executar em simultâneo em sistemas multiprocessador.

B) poderão executar com cautela e rapidez.

C) não poderão executar em simultâneo.

18. Um programa não deve ter mais de uma zona crítica.

A) Sim, uma só zona crítica é o máximo admissível.

B) Não há limite para o número de zonas críticas mas o seu número deverá ser o menor possível.

C) Não há limite para o número de zonas críticas, mas só no caso de se usarem monitores.

19. O problema do produtor-consumidor não pode ser resolvido por simples mutexes.

A) Pois não, porque para além de se garantir exclusão mútua ao meio de armazenagem de itens, é necessário controlar-se a produção e o consumo de itens.

B) Nada disso: é claro que qualquer problema genérico de sincronização pode ser resolvido apenas com mutexes!

C) A veracidade da afirmação depende do tipo de itens em causa.

20. A instrução `TestAndSetLock` (TSL), que garante de uma forma atómica a leitura de uma variável em memória e a alteração do seu valor, é implementada num sistema como se mostra junto.
De entre as opções seguintes, escolha a que correctamente usa a função `TSL()` para proteger uma zona crítica de código.
[NOTA: na inicialização do sistema, `int flag = 0;` e `scheduler()` invoca o escalonador, cedendo o uso do processador a outro processo.]

```
int TSL (int *ptr, int new) {
   int old = *ptr;
      *ptr = new;
      return old;
}
```

A)
```
while (TSL(&flag, 1) != 0) scheduler();
   /* ... Critical Region ... */
   flag = 1;
```

B)
```
while (TSL(&flag, 0) != 0) scheduler();
   /* ... Critical Region ... */
   flag = 0;
```

C)
```
while (TSL(&flag, 1) != 0) scheduler();
   /* ... Critical Region ... */
   flag = 0;
```

21. Considere o seguinte programa, muito simplificado, em que se pretende que o *thread* principal termine só depois do *thread* `func()` ter efectuado o trabalho, `work()`.

```
int done = 0;   // li.1

void *func(void *a) {
     work();    // takes some secs
     done = 1;  // li.2
     return(NULL);
}
```

```
int main() {
     pthread_t tid;
     ;     // li.3
     pthread_create(&tid, NULL, func, NULL);
     while (done == 0) ;  // li.4
     return 0;
}
```

As linhas assinaladas por um comentário com '`li.n`' (onde `n` é 1, 2, 3 e 4) poderão ser substituídas por código que altere o programa de forma a evitar a espera activa. De entre as alíneas seguintes, escolha a opção de código de substituição das linhas que realmente cumpre (correctamente) o requisito

a) usando `pthread_join()`:

A)
```
li.1: comentar linha
li.2: comentar linha
li.3: int res;
li.4: pthread_join(&tid, res);
```

B)
```
li.1: manter linha
li.2: comentar linha
li.3: comentar linha
li.4: while (done == 0)
        pthread_join(&tid, NULL);
```

C)
```
li.1: comentar linha
li.2: comentar linha
li.3: comentar linha
li.4: pthread_join(tid, NULL);
```

b) não usando `pthread_join()`:

A)
```
li.1: pthread_mutex_t done;
li.2: pthread_mutex_unlock(&done);
li.3: pthread_mutex_init(&done, NULL);
li.4: pthread_mutex_lock(&done);
```

B)
```
li.1: sem_t done;
li.2: sem_post(&done);
li.3: sem_init(&done, 0, 0);
li.4: sem_wait(&done);
```

C)
```
li.1: pthread_mutex_t mt; pthread_cond_t done;
li.2: pthread_cond_signal(&done);
li.3: pthread_mutex_init(&mt, NULL);
        pthread_cond_init(&done, NULL);
li.4: pthread_cond_wait(&done, &mt);
```

22. Uma das estratégias para se lidar com encravamentos 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 pedidos de memória aos processos dos utilizadores.

23. 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.

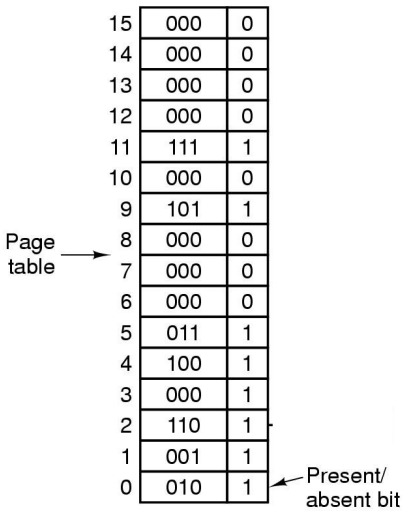
24. Um sistema de memória virtual paginada tem a vantagem de

A) evitar a fragmentação externa da memória física.

B) permitir ao programador aplicacional um maior controlo sobre a estrutura de memória física.

C) melhorar o acesso à memória física pela diminuição da utilização de *swapping*.

25. Na figura junto está representada uma Tabela de Páginas muito simples, usada nas folhas de apoio às aulas teóricas, e que serviu para exemplificar as operações com endereços efectuadas pela Unidade de Gestão de Memória (MMU).



a) Indique o valor de saída da MMU quando um programa pretende aceder ao endereço 5 : 9 (*page : offset*).

A) 3 : 9 (*frame : offset*)

B) 9 : 3 (*frame : offset*)

C) 9 : 9 (*frame : offset*)

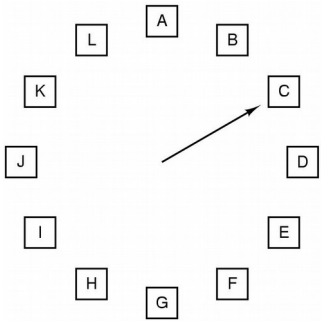
b) Indique a quantidade de memória física instalada, sabendo que cada página tem o tamanho de 4 KiB.

A) 16 KiB

B) 32 KiB

C) 48 KiB

26. O algoritmo de relógio representado na figura junto, também conhecido como "FIFO, second chance", foi apresentado nas folhas de apoio às aulas teóricas, no capítulo de memória virtual paginada.



a) Este algoritmo é um dos que podem ser usados na

A) identificação de um quadro (*frame*) de memória física que irá receber nova página do disco.

B) identificação de uma página de memória *cache* que irá substituir uma página pouco usada do disco.

C) identificação de uma página em disco que irá ser colocada na memória física.

b) Suponha que o ponteiro mostrado avança no sentido horário (!) e que em C o bit de referência (R) tem o valor 1 e o bit de Modificação (M) tem o valor 0. Nessa situação, quando o algoritmo é invocado, a identificação referida na alínea a) vai recair

A) em C.

B) em D.

C) (ainda nada se pode dizer).

27. O controlo da informação sobre nomes e permissões de acesso a dispositivos de Entrada/Saída (I/O) é atribuída

A) ao *software* de sistema do controlador dos dispositivos.

B) ao *software* de sistema da camada independente dos dispositivos.

C) ao *software* de sistema ligado directamente aos dispositivos.

28. O *device driver* de um dispositivo Entrada/Saída (I/O) é o *software*

A) que vem em ROM junto com o dispositivo.

B) que faz a ligação entre o núcleo sistema operativo e o dispositivo.

C) que faz a ligação entre o programa do utilizador e o dispositivo.

29. Muitos dos pedidos a dispositivos de Entrada/Saída (I/O) são bloqueantes por, normalmente, não poderem ser atendidos com a rapidez necessária à operação do processador. Por isso

A) é importante que se faça uso de transferências directas para a memória (DMA) via *threads*.

B) é importante que o processo requisitante faça uma espera activa até chegar a resposta.

C) é importante que uma interrupção avise o sistema quando a resposta chegar, a fim de o processo requisitante ser acordado.

30. O modo de controlo de teclado que aguarda que uma quantidade de informação seja introduzida (tipicamente, até à detecção de [ENTER]) antes de a processar e passar ao resto do sistema, denomina-se

A) *canonical* (ou *cooked*).

B) *non-canonical* (ou *raw*).

C) *bufferized* (ou *customized*).

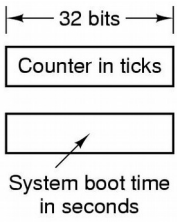
31. Em cada *tick* do relógio interno de um computador

A) uma interrupção é gerada e um conjunto de tarefas de sistema efectuadas.

B) o processador é retirado a um processo de utilizador e entregue a outro processo.

C) uma interrupção é enviada a todos os dispositivos de Entrada e Saída (I/O) que têm operações em curso.

32. O esquema junto mostra a maneira como o relógio interno de um dado computador é mantido. Sendo *C* o valor do contador superior (*ticks*) e *S* o valor do contador inferior (segundos) num dado instante, como pode o sistema fornecer a data (e hora) civil, nesse instante e com a máxima precisão, sabendo que *R* é o nº de *ticks* por segundo?



A) Data (e hora) em segundos: $S + C * R$

B) Data (e hora) em segundos: *Epoch time* + $S + C / R$

C) Data (e hora) em segundos: $S + C / R$

33. Num disco de estado sólido (SSD) do tipo NAND Flash, diferentes operações de acesso demoram tempos diferentes. Indique qual da seguintes alíneas representa, de forma aproximada, o estado tecnológico actual:

A) leitura de página: ~ 100 us ; escrita de página (já apagada) ~ 10 us ; apagamento de bloco ~ 100 us

B) leitura de página: ~ 10 us ; escrita de página (já apagada) ~ 100 us ; apagamento de bloco ~ 10 us

C) leitura de página: ~ 10 us ; escrita de página (já apagada) ~ 100 us ; apagamento de bloco ~ 1000 us

34. O executável compilado a partir do código simplificado (e sem detecção de erros) abaixo, `prog1`, deverá ir lendo o texto ASCII de um ficheiro cujo nome é passado como 1º argumento da linha de comando e escrevendo o resultado num novo ficheiro cujo nome é passado como 2º argumento. Um exemplo de utilização do `prog1` seria:

```
shell$ prog1 fich fich.maiusc
```

```
void maiusc(int fd1, int fd2) {
   char c;
   while (read(fd1, &c, 1)) > 0) {
      c = (char) toupper(c);
      write(fd2, &c, 1);
   }
}
```

```
int main(int argc, char *argv[]) {
   int fd1 = open(argv[1], O_RDONLY)); // li.1
   int fd2 = open(argv[2], O_WRONLY | O_TRUNC |
O_CREAT, 0644); // li.2
   maiusc(fd1, fd2); // li.3
   return 0;
}
```

a) Por substituição das linhas assinaladas por um comentário com '`li.n`' (onde `n` é 1, 2), altere o programa de forma a que o novo programa compilado, `prog2`, agora leia texto da entrada padrão (*stdin*) e coloque na saída padrão (*stdout*) o correspondente texto em maiúsculas. Um exemplo de utilização do `prog2` seria:

```
shell$ prog2 < fich > fich.maiusc
```

De entre as alíneas seguintes, escolha a opção de código de substituição das linhas referidas que cumpra (correctamente) o novo requisito.

A)
```
int fd1 = STDIN_FILENO; // li.1
int fd2 = STDOUT_FILENO; // li.2
```

B)
```
int fd1 = open(stdin, O_RDONLY)); // li.1
int fd2 = open(stdout, O_WRONLY | O_TRUNC | O_CREAT, 0644); // li.2
```

C)
```
int fd1 = stdin; //li.1
int fd2 = stdout; // li.2
```

b) Suponha agora que o executável `prog2` está disponível para utilização no directório corrente e pretende-se construir um novo programa cujo executável correspondente, `prog3`, reproduza o funcionamento do `prog1` inicial, mas por invocação de `prog2`! Um exemplo de utilização do `prog3` seria:

```
shell$ prog3 fich fich.maiusc
```

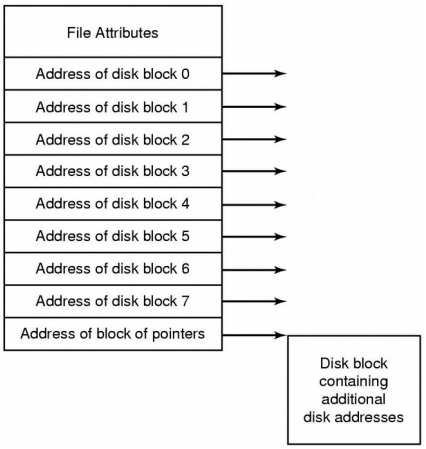
De entre as alíneas seguintes, escolha a opção de código de substituição da linha 3 (li.3) que cumpra (correctamente) o novo requisito.

A) `dup2 (fd1, stdin); dup2 (fd2, stdout); execlp("./prog2", NULL, NULL); // li.3`

B) `dup2 (STDIN_FILENO, fd1); dup2 (STDOUT_FILENO, fd2); execlp("./prog2", "prog2", NULL); // li.3`

C) `dup2 (fd1, STDIN_FILENO); dup2 (fd2, STDOUT_FILENO); execlp("./prog2", "prog2", NULL); // li.3`

35. Considere um sistema de ficheiros com a implementação de *i-nodes* exemplificada na figura junto.



a) Supondo que nesse sistema cada bloco tem 4096 B e cada endereço de bloco de disco usa 4 B, o tamanho máximo que um ficheiro pode ter é, em KiB:

A) 4128 KiB.

B) 4132 KiB.

C) 16416 KiB

b) No campo "File Attributes" costuma estar incluído:

A) o nome do ficheiro.

B) o tamanho do ficheiro.

C) a data da criação do ficheiro.

36. Em sistemas Unix, os nomes de ficheiros que contêm "extensão" (`.txt`, `.c`, etc.) determinam a maneira como esses ficheiros são tratados no sistema de ficheiros.

A) Sim, pois os *i-nodes* correspondentes têm um campo específico para a extensão.

B) Não, não há provisão especial para esses ficheiros.

C) Talvez, se nos directórios correspondentes se usar o atributo relativo à extensão.

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

A) consultar a entrada correspondente no directório onde o ficheiro está armazenado.

B) consultar a entrada correspondente no directório onde o ficheiro está listado.

C) consultar o primeiro bloco do ficheiro, para ver onde está a correspondente entrada do directório.

38. Considere o seguinte excerto simplificado de código, que respeita à parte servidor de uma aplicação do género da usada no 2º mini-projecto. Suponha que não se dão situações inesperadas de erro (além do que estiver previsto nas perguntas).
Lembra-se que o servidor esperava num FIFO público por pedidos de clientes e respondia a cada pedido através de um FIFO privado preparado pelo cliente requisitante.

```
...    // FIFO publico é criado e aberto para leitura: pubfifo
while (read(pubfifo, &msgreq, sizeof(msgreq)) > 0) {
       ... /* processa pedido msgreq e prepara resposta msgreply para o FIFO privado */
       privfifo = open(privatefifoname, O_WRONLY);
       write(privfifo, msgreply, sizeof(msgreply));
       }
...    // FIFO publico é fechado e eliminado
```

a) De entre as opções seguintes escolha a que tem o excerto de código para o cliente que permitirá o bom funcionamento da aplicação.

A)
```
... // FIFO publico é aberto para escrita: pubfifo
char *clientfifoname = ... // aponta para nome privado
mkfifo(clientfifoname, 0666);
int clientfifo = open(clientfifoname, O_RDONLY);
... /* constroi pedido request */
write(pubfifo, request, sizeof(request));
read(clientfifo, &answer, sizeof(answer));
... /* processa resposta answer */
```

B)
```
... // FIFO publico é aberto para escrita: pubfifo
char *clientfifoname = ... // aponta para nome privado
... /* constroi pedido request */
write(pubfifo, request, sizeof(request));
mkfifo(clientfifoname, 0666);
int clientfifo = open(clientfifoname, O_RDONLY);
read(clientfifo, &answer, sizeof(answer));
... /* processa resposta answer */
```

C)
```
... // FIFO publico é aberto para escrita: pubfifo
char *clientfifoname = ... // aponta para nome privado
mkfifo(clientfifoname, 0666);
... /* constroi pedido request */
write(pubfifo, request, sizeof(request));
int clientfifo = open(clientfifoname, O_RDONLY);
read(clientfifo, &answer, sizeof(answer));
... /* processa resposta answer */
```

b) Se se desejasse que a aplicação permitisse uma interação bidirectional privada (uma "conversa" privada) entre cada cliente e o servidor, poder-se-ia usar os FIFO privados para isso?

A) Sim, porque os canais já estavam abertos e eram privados.

B) Não, porque os FIFO privados só passam mensagens num só sentido.

C) Talvez, se se alterasse a especificação de abertura dos FIFO privados (parâmetros adicionais no `open()`).

39. Em muitos dispositivos de base computacional (i.e., que incluem um processador) quer-se um sistema operativo simples porque

A) pode não haver necessidade de ligar à Internet.

B) pode haver limitação de memória física.

C) pode haver necessidade de controlo de custos.

40. A protecção de um sistema computacional contra canais camuflados (*covert channels*) pode ser conseguida com o auxílio do sistema operativo.

A) Sim, é certo, porque o sistema operativo pode detectar a existência de tais canais.

B) Talvez possa ser mitigada, caso o administrador faça uma configuração adequada do sistema.

C) Não, este é um tipo de ameaça que é quase impossível de evitar, em geral.

41. Uma Matriz de Protecção é controlada pelo sistema operativo e permite definir

A) as autorizações dos utilizadores no acesso aos recursos partilhados.

B) a maneira como os utilizadores se autenticam no sistema.

C) os canais de comunicação confidencial entre processos.

42. Usamos Unix/Linux na disciplina porque, entre outras coisas:

A) está fácil e gratuitamente acessível e ilustra etapas fundamentais da história dos sistemas operativos.

B) é o único sistema que permite uma interacção em modo texto com o utilizador.

C) há nas aulas práticas trabalhos em que se precisa alterar o código de funções específicas do sistema.

Respostas e Comentários noutra folha, que é fornecida.