Sistemas Operativos (2012/13) **Exame: Sem consulta**
20 de Junho de 2013 **Duração: 120 min**

1- [8v] Em cada alínea desta pergunta deverá indicar se a afirmação feita é verdadeira ou falsa e explicar a sua resposta como exemplificado em seguida.
No caso da afirmação ser **verdadeira, deverá explicar a razão de assim ser**. Por exemplo, dada a afirmação:

> A instalação dum módulo no *kernel* do Linux **não** pode ser realizada por um utilizador comum.

uma possível resposta é:

> Verdadeiro. Após a sua instalação um módulo executa em modo priveligiado, podendo executar qualquer ação. Assim, se qualquer utilizador pudesse instalar um módulo, poderia executar qualquer ação incluindo, p.ex., ler o conteúdo de ficheiros de outros utilizadores.

No caso da afirmação ser **falsa, deverá alterar a afirmação de forma a torná-la verdadeira de forma não trivial e explicar a razão de assim ser**. Por exemplo, dada a afirmação:

> A instalação dum módulo no *kernel* do Linux pode ser realizada por qualquer utilizador.

uma possível resposta é:

> Falso. Um utilizador comum não pode instalar um módulo do *kernel* do Linux. Após a sua instalação um módulo executa em modo priveligiado, podendo executar qualquer ação. Assim, se qualquer utilizador pudesse instalar um módulo, poderia executar qualquer ação incluindo, p.ex., ler o conteúdo de ficheiros de outros utilizadores.

Como ilustrado por estes exemplos, a resposta não deverá exceder 3 ou 4 frases (uma resposta demasiado longa poderá ser penalizada).

1.1- Em sistemas operativos com um único utilizador o processador deverá suportar apenas o modo de execução priveligiado.

1.2- A única vantagem de multiprogramação em SO com múltiplos utilizadores é permitir usar os recursos dum computador duma forma mais eficiente.

1.3- Normalmente as funções das bibliotecas de *threads* implementados ao nível do utilizador são mais eficientes do que as das bibliotecas de *threads* implementados ao nível do *kernel*.

1.4- Tanto *locks/mutexes* como semáforos permitem evitar *race conditions* quando usados de forma apropriada.

1.5- A partilha do CPU em sistemas multiprocesso pode conduzir a bloqueio mútuo (*deadlock*).

1.6- Com E/S programada *busy-waiting* é inevitável.

1.7- Num sistema com n discos SATA o SO precisa de n *device drivers*, um por disco.

1.8- Quanto maior é a *Translation Lookaside Buffer (TLB)* menos relevante é a quantidade de memória num computador para o desempenho de memória virtual.

1.9- A API das chamadas ao sistema de acesso a ficheiros de Unix/Linux (`read()` e `write()`) pressupõem um acesso sequencial.

1.10- A tecnologia SSD não terá qualquer efeito na conceção de sistemas operativos, pois a sua interface ao nível do HW é semelhante à dos "hard disks", e.g. SATA.

2- [3v] Sobre o projecto de desenvolvimento dum *device driver* para a porta série em Linux.

2.1- Um dos passos da inicialização do seu *device driver* consistiu na invocação da função:

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
```

Assuma que se esqueceu de invocar esta função. Quais os problemas que daí adviriam? Manifestar-se-iam na invocação da chamada ao sistema `open()` ou nas chamadas ao sistema de transferência de dados, p.ex. `read()`? Justifique.

2.2- Considere o projeto. Explique de forma sumária a **implementação** de **uma** das seguintes funções da estrutura `struct fileops`:

1. Função `read` em modo *polling*, suportando a leitura de mais de 1 carácter por chamada ao sistema.
2. Função `write` em modo de interrupção.

IMP.: A cotação desta alínea depende da função: 80% para `read`, e 100% para `write`.

3- [2v] Considere a implementação dum *device driver* (DD) para um dispositivo de entrada, num sistema operativo para sistemas multiprocessador/multicore. A comunicação entre o *interrupt handler* (IH) e (a parte restante d)o DD é realizada através duma fila usando-se um *spin lock* para garantir acesso em exclusão mútua a essa fila. O segmento de código seguinte contém a parte relevante da comunicação descrita. Assuma que as funções invocadas fazem o que os seus nomes sugerem e que não têm *bugs*. Assuma ainda que os argumentos usados são os esperados e que as funções `enter()` e `remove()` não usam qualquer mecanismo/primitiva de sincronização adicional.

DD

```
...
A: spin_lock(&q->l);
B: buf = remove(q);
C: spin_unlock(&q->l);
...
```

IH

```
...
1: spin_lock(&q->l);
2: enter(q, buf);
3: spin_unlock(&q->l);
...
```

3.1- Ilustre, apresentando uma sequência de instruções, como a comunicação entre o DD e o IH pode conduzir a um problema. Use os números indicados no início de cada linha, e designe os *threads* envolvidos pelos acrónimos DD e IH.

Deverá ainda: i) indicar o estado da fila no início dessa sequência; ii) descrever sucintamente os eventos que causam que 2 instruções consecutivas dessa sequência seja executadas por *threads* diferentes; iii) explicar sumariamente porque esta sequência é problemática.

3.2- Explique como poderia resolver este problema recorrendo a um mecanismo/primitiva de sincronização **adicional**. Apresente os segmentos de código modificados resultantes.

3.3- Explique porque razão o segmento de código ilustrado acima estaria correto se fosse usado para a comunicação entre 2 *threads* normais num sistema multiprocessador/multicore (e não entre o DD e o seu IH).

4- [1.5v] Sobre memória virtual paginada. Considere um espaço de endereçamento virtual paginado de 32 bits e 2 alternativas para a sua tabela de páginas:

1. Tabela com um único nível e páginas de 1 MiByte,
2. Tabela com dois níveis, páginas de 4 KiByte, em que se usa 10 bits para cada um dos níveis da tabela.

Em ambos os casos, cada elemento de tabela de páginas ocupa 4 bytes.

4.1- Assumindo que em média cada processo ocupa apenas 5% do seu espaço de endereçamento virtual, qual é o espaço de memória usado em média para a tabela de páginas de cada processo, para cada uma das duas alternativas.

4.2- Dê uma vantagem (diferente do tamanho de memória usado para a tabela de páginas) para cada uma das duas alternativas indicadas.

5- [1.5v] Sobre sistemas de ficheiro tipo Unix.

5.1- O utilitário cp cria uma cópia dum ficheiro cujo nome lhe é dado como primeiro argumento atribuindo ao novo ficheiro o nome especificado no seu segundo argumento. Indique, justificando, as estruturas do sistema de ficheiros mantidas no disco que serão alteradas pela execução deste utilitário.

5.2- Descreva como um "crash" do sistema durante a execução deste utilitário pode deixar estas estruturas de dados num estado inconsistente.

6- [4v] Pretende-se desenvolver uma aplicação *multithreaded* para contagem do número de palavras em múltiplos ficheiros. Assuma que dispõe duma função, de nome `wc()`, a qual recebe como argumento o nome de um ficheiro de texto e retorna o nº de palavras nele contido.
O programa deverá fazer o seguinte:

- ler da entrada padrão (*stdin*), linha a linha, nomes de ficheiros de texto (até surgir EOF - *end of file*);
- para cada nome, invocar um *thread* que deverá executar a função `wcountf()`, a qual recebe o nome dum ficheiro como argumento, invoca a função `wc()`, descrita mais acima, e antes de terminar imprime na saída padrão (*stdout*) o nome do ficheiro de texto e o nº de palavras nele contido

6.1- Escreva este programa, garantindo que cada *thread* diferente do principal termina independentemente deste último.
Nota: suponha que todas as chamadas a rotinas de sistema serão bem sucedidas(!).
Dica: se tem dificuldade em implementar a leitura do nome dos ficheiros como descrito acima, especifique o protótipo duma função que execute essa tarefa, assuma que já está implementada e invoque-a na parte restante do seu código. Posteriormente, se tiver tempo, poderá tentar implementar essa função. (A sua implementação não vale mais de 20% da cotação desta alínea.)

6.2- Escreva uma nova versão do mesmo programa que difere da anterior no facto do nº de *threads* do programa não poder exceder o valor $n+1$ onde n é o único argumento da linha de comando desta versão do programa.
Dica: Quando o nº de *threads* for o máximo, o *thread* principal deverá esperar de forma não ativa. Quando um *thread* de contagem estiver para terminar, deverá notificar o principal para que este possa criar novo *thread* de contagem.
Nota: suponha que todas as chamadas a rotinas de sistema serão bem sucedidas(!).

Protótipos

Processos

```
pid_t fork(void);
int execve(const char *filename, char *const argv[], char *const envp[]);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
void _exit(int status);
WIFEXITED(); // macro
WEXITSTATUS(); // macro
```

Threads

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

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);
void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
        prot: PROT_NONE, PROT_EXEC, PROT_READ, PROT_WRITE
        flags: MAP_FIXED, MAP_SHARED, MAP_PRIVATE
int munmap(void *start, size_t length);
int msync(void *start, size_t length, int flags);
off_t lseek(int fildes, off_t offset, int whence);
       whence: SEEK_SET, SEEK_CUR, SEEK_END
```