Sistemas Operativos (2010/11) **Exame Sem Consulta (A)**
9 de Junho de 2011 **Duração: 105+15 min**

Nome:

1- [8v] Em cada alínea desta pergunta deverá escolher a afirmação mais completa que é verdadeira. Indique-a no espaço reservado para o efeito nesta folha, a qual deverá ser entregue no final.

1.1- Que operações exigem execução com nível de privilégio elevado?

A O processamento duma interrupção.
B A compilação dum módulo do núcleo em Linux.
C O processamento duma interrupção e a compilação do módulo do núcleo em Linux.

Resposta:

1.2- Sobre multiprogramação e paralelismo.

A Paralelismo real só é possível em computadores com processador *multicore* e/ou múltiplos processadores.
B O uso de multiprogramação só tem interesse quando o sistema operativo suporta múltiplos utilizadores.
C As duas afirmações anteriores são verdadeiras.

Resposta:

1.3- Sobre *threads*.

A *Threads* dum mesmo processo partilham a *stack*, mas *threads* de processos diferentes não.
B *Threads* dum mesmo processo partilham o mesmo espaço de endereçamento, mas *threads* de processos diferentes não.
C Nenhuma das afirmações anteriores é verdadeira.

Resposta:

1.4- Sobre escalonamento de processos.

A O escalonamento de processos é um caso particular da gestão de recursos num SO.
B Algoritmos de escalonamento baseados em prioridades podem conduzir à míngua (*starvation*).
C As duas afirmações anteriores são verdadeiras.

Resposta:

1.5- Sobre mecanismos de *hardware* para suporte à exclusão mútua.

A Estes mecanismos não são por si só suficientes para implementar de sincronização de processos sem espera activa(*busy waiting*)
B Num sistema multiprocessador sem instruções *read-modify-write* atómicas, pode usar-se a inibição de interrupções desde que o SO escalone processos que executam *threads* dum mesmo conjunto de secções críticas no mesmo processador
C As duas afirmações anteriores são verdadeiras

Resposta:

1.6- Sobre *deadlocks* (bloqueio mútuo).

A A preempção de recursos raramente é uma solução satisfatória para resolver "deadlocks".
B Se num dado estado de alocação de recursos não há "deadlock", diz-se que o estado é seguro.
C "Deadlocks" ocorrem apenas no contexto de partilha de recursos.

Resposta:

1.7- Sobre sistemas de entrada/saída.

A Em Linux todos os dispositivos de E/S são acedidos através dum ficheiro do directório `/dev`

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

C As duas afirmações anteriores são verdadeiras.

Resposta:

1.8- Sobre memória virtual (MV) paginada.

A Memória virtual permite que o tamanho dum processo seja maior do que a quantidade de memória física disponível num computador.

B Em computadores em que o espaço de endereços físico seja igual ou superior ao espaço de endereços virtual não há vantagem em usar memória virtual paginadas.

C Nenhuma das afirmações anteriores é verdadeira.

Resposta:

1.9- Sobre *Table Look Aside Buffers (TLB)s*.

A A taxa de *TLB misses* imediatamente após a comutação de processos é tipicamente superior à mesma taxa antes dessa comutação ocorrer.

B Uma *TLB miss* normalmente é mais penalizadora do desempenho dum processo do que uma *page fault*.

C Quanto maior é a quantidade de memória num computador, menos relevante é a TLB para o desempenho de MV.

Resposta:

1.10- Sobre sistemas de ficheiros em Unix/Linux.

A Em Unix todos os ficheiros suportam acesso aleatório.

B As chamadas ao sistema `read()` e `write()` em Unix pressupõem que normalmente o acesso a ficheiros é sequencial.

C Em Unix, directórios residem em disco, mas é concebível que, por razões de desempenho, em outros sistemas operativos residam em memória RAM.

Resposta:

2- **[2v]** Sobre o projecto.

2.1- Admita que no desenvolvimento dum *device driver* como módulo do núcleo se esqueceu de invocar a função:

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name)
```

na função de inicialização do módulo. Explique os problemas que daí podem advir.

2.2- Uma estrutura de dados essencial no desenvolvimento de *device drivers* em Linux é a estrutura

```
struct file_operations {
        module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
        [...]
};
```

Explique i) para que serve; ii) quando e como deve ser inicializada; e iii) quando é que os seus membros são acedidos.

3- **[2v]** Responda a cada uma das perguntas seguintes.

3.1- A probabilidade de ocorrência duma *race-condition* será menor se o SO usar escalonamento sem preempção (vs. com preempção)? Explique.

3.2- Explique como é que um sistema de memória virtual paginada protege um processo contra o acesso indevido de outros processos à sua memória.

4- **[2v]** Considere os seguintes segmentos duma solução para o problema do *bounded-buffer*:

Produtor

```
A: pthread_mutex_lock( &bbuf_p->l );
B: while( bbuf_p->cnt == BUF_SIZE ) {
C:   pthread_mutex_unlock( &bbuf_p->l );
D:   sleep(bbuf_p);
E:   pthread_mutex_lock( &bbuf_p->l );
F: }
G: enter(bbuf_p, (void *)req_p);
H: if( bbuf_p->cnt == 0 )
I:    wakeup(bbuf_p);
J: pthread_mutex_unlock( &bbuf_p->l );
```

Consumidor

```
 1: pthread_mutex_lock( &bbuf_p->l );
 2: while( bbuf_p->cnt == 0 ) {
 3:   pthread_mutex_unlock( &bbuf_p->l );
 4:   sleep(bbuf_p);
 5:   pthread_mutex_lock( &bbuf_p->l );
 6: }
 7: req_p = (req_t *)remove( bbuf_p );
 8: if( bbuf_p->cnt == BUF_SIZE -1 )
 9:    wakeup(bbuf_p);
10: pthread_mutex_unlock( &bbuf_p->l )
```

onde `sleep()` é uma chamada ao sistema que bloqueia o *thread* que a executa até que outro *thread* invoque a chamada ao sistema `wakeup()` com o mesmo argumento.

Mostre através duma sequência de instruções que este código pode dar origem a uma *race condition*. (Use os números indicados no início de cada linha.) Deverá ainda i) indicar o estado do *bounded buffer* no início dessa sequência; ii) descrever sucintamente os eventos que determinam que 2 instruções consecutivas dessa sequência sejam executadas por *threads* diferentes; iii) descrever de forma sumária a *race condition* identificada.
IMP. Assuma que as funções `enter()` e `remove()` inserem e removem, respectivamente, um elemento no *bounded buffer*, estão correctamente implementadas e não usam qualquer mecanismo de sincronização.

5- **[2v]** Considere o seguinte segmento de código que copia o conteúdo dum ficheiro para outro ficheiro.

```
/* File copy program. Minimal error checking */
int main(int argc, char *argv[]) {
    int in_fd, out_fd, rd_cnt, wr_cnt;
    char buf[BUF_SIZE];
    if (argc != 3) exit(1);            /* syntax error */
    in_fd = open(argv[1], O_RDONLY);   /* open source file */
    if (in_fd < 0) exit(2);
    out_fd = open(argv[2], O_WRONLY | O_CREAT, S_IRWXU );  /* create sink file */
    if (out_fd < 0 ) exit(3);           /* error in open */
    while (TRUE) {          /* loop until done, or an error */
        rd_cnt = read(in_fd, buf, BUF_SIZE); /* read from source */
        if (rd_cnt <= 0) break;    /* end of file, or error */
        wr_cnt = write_buf(out_fd, buf, rd_cnt); /* write block read */
        if (wr_cnt != 0) exit(4);        /* error writing */
    }
    close(in_fd);                      /* close files */
    close(out_fd);
    if (rd_cnt == 0)                   /* no error on last read */
        exit(0);
    else                               /* error on last read */
        exit(5);
}
```

Escreva a função:

```
int write_buf(int fd, void *buf, int count)
```

a qual deve retornar apenas após ter escrito no ficheiro com descritor `fd` todos os `count` *bytes* no *buffer* `buf`, ou após a ocorrência dum erro. `write_buf()` deverá retornar o número de *bytes* **por escrever**.

6- [4v] Considere o seguinte segmento de código dum programa com múltiplos *threads* para calcular a média dum vector.

A função `partial_aver()` é usada para calcular a média duma parte do vector e toma como argumento um endereço da estrutura do tipo `struct targ`. A função `average()` é usada para o cálculo da média a partir das médias partiais.

```
 1: #define ARRAY_SIZE 10000000
 2: #define NT  10
 3:
 4: pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
 5: pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
 6: int arr[ARRAY_SIZE];    /* array of values */
 7: int pavg[NT];           /* array for partial averages */
 8: int avg = 0;            /* average to compute */
 9: int done = 0;           /* number of partial averages computed */
10:
11: struct targ {
12:    int thrd_no;        /* number of thread */
13:    int len;            /* length of sub-array */
14: };
15:
16: void *partial_aver(void *arg) {
17:     int i, sum;
18:     int thrd_no = ((struct targ *)arg)->thrd_no;
19:     int len = ((struct targ *)arg)->len;
20:
21:     for(i = thrd_no*len, sum = 0; i < (thrd_no+1)*len; i++)
22:         sum += arr[i];
23:     pavg[thrd_no] = sum/len;
24:     done++;
25:     pthread_exit(NULL);
26: }
27:
28: void *average(void *arg) {
29:     int i, sum;
30:
31:     while( done != NT  ) {
32:         sleep(1);
33:     }
34:     for(i = 0, sum = 0 ; i < NT; i++)
35:         sum += pavg[i];
36:     avg = sum/NT;
37:     pthread_exit(NULL);
38: }
39:
40: int main() {
41:    [...]        /* code not relevant */
42:    compute_average();
43:    printf("average = %d\n", avg);
44:    return 0;
45: }
```

6.1- Escreva a função `compute_average()` a qual deverá criar `NT` *threads* para cálculo das médias parciais e um *thread* para cálculo da média. Este último deverá ser criado antes dos anteriores. `compute_average()` deverá ainda esperar pela terminação de todos os *threads* que criar.

6.2- Usando o *mutex* já declarado e inicializado, proteja de forma apropriada as secções críticas deste programa de forma a maximizar a concorrência.

6.3- Este programa usa espera activa. Usando a variável de condição já declarada e inicializada, faça as alterações necessárias para eliminar a espera activa.

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);
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
```