Sistemas Operativos (2011/12) **Exame: Sem consulta**
8 de Junho de 2012 **Duração: 120 min**

Nome:

1- [8v] Em cada alínea desta pergunta deverá indicar se a afirmação feita é verdadeira ou falsa e justificar a sua resposta. A justificação não deverá exceder 2 ou 3 frases (uma resposta demasiado longa poderá ser penalizada).

1.1- A instalação dum módulo no *kernel* do Linux exige execução com nível de privilégio elevado.

1.2- A chamada ao sistema `execve()` retorna 0 se bem sucedida e -1 caso contrário.

1.3- Se as aplicações não bloqueassem em operações de E/S não havia vantagem em suportar multiprogramação.

1.4- Numa implementação híbrida o bloqueio dum *thread* devido a uma *page fault* nunca impede por si só a execução de qualquer dos restantes *threads*.

1.5- Quando se usa semáforos como *locks* tipicamente inicializa-se o semáforo com o valor 0.

1.6- Operações de E/S assíncronas são especialmente úteis no desenvolvimento de aplicações com múltiplos *threads*.

1.7- Num sistema de memória virtual todas as páginas dum processo estão ou na memória principal ou na área de *swap*.

1.8- Cada elemento da tabela não invertida de páginas (*page table entry*) inclui tipicalmente o nº de página e o nº de *frame*.

1.9- Em Unix/Linux diretórios são implementados como ficheiros sendo comum usar as chamadas ao sistema `read()` e `write()` para aceder ao seu conteúdo.

1.10- O uso de múltiplas zonas em discos aumenta a sua capacidade mas prejudica o desempenho dos sistemas de ficheiros.

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

2.1- Muito provavelmente, o seu *device driver* invoca:

```
struct resource *request_region(unsigned long first, unsigned long n, const char *name)
```

na função de inicialização do módulo, mas poderia tê-la invocada quando do "primeiro" `open()`. Qual é a vantagem desta última aproximação? Justifique.

2.2- Considere a seguinte funcionalidade que consta do documento final de especificação do projecto:

1. Qualquer dos pontos 3.2 (`ioctl`) a 3.8 (FIFOs), excepto 3.6
2. Ponto 3.6, implementação da função `poll` da estrutura `struct fileops`.
3. Transferência de dados entre a UART em modo de interrupção e o processo da aplicação, em `read()` ou em `write()`.

Descreva de forma sumária a implementação de uma delas à sua escolha. A quotação desta alínea depende da funcionalidade descrita como se segue: 75% para uma funcionalidade do ponto 1, 90% para a funcionalidade do ponto 2, e 100% para uma das duas funcionalidades do ponto 3.

3- [1.5v] Um levantamento de requisitos dum sistema de ficheiros determinou os seguintes fatores e respetivos pesos:

Factor	**Peso**
Acesso aleatório aos ficheiros	0,5
Acesso sequencial aos ficheiros	0,3
Utilização da capacidade do disco	0,2

Determine qual das 3 estruturas típicas usadas para manter informação sobre a localização dos blocos dum ficheiro:

1. Alocação contígua
2. Listas ligadas
3. Tabelas multi-nível com apontadores

é a mais adequada para este caso.
Note que cada estrutura deverá ser classificada com base nos fatores acima numa escala de 1 a 3, correspondente à posição relativa de cada uma das estruturas, sendo 3 para a melhor e 1 para a pior. Assim, p.ex. se uma das estruturas tiver as seguintes posições relativas (pior, melhor, melhor), a sua classificação será:

$$1 \times .5 + 3 \times .3 + 3 \times .2 = 2$$

4- [2v] Considere a seguinte solução baseada em programação para o problema da secção crítica com 2 processos apenas:

```
1: void enter(int id, int *turn) {
2:     while( *turn != id );
3: }
```

```
A: void leave(int id, int *turn) {
B:     *turn = 1 - id;
C: }
```

Assume-se:

- que uma operação de leitura/escrita de uma variável inteira em memória é atómica;
- que cada um dos processos tem um identificador com valor ou 0 ou 1;
- que cada secção crítica tem uma variável inteira que é inicializada com o valor 0;
- que um processo antes de entrar numa secção crítica invoca `enter()` passando-lhe o seu identificador e o endereço da variável associada à secção crítica como 1º e 2º argumentos, respectivamente;
- que um processo depois de executar a secção crítica invoca `leave()` passando-lhe o seu identificador e o endereço da variável associada à secção crítica como 1º e 2º argumentos, respectivamente;

Esta solução satisfaz os requisitos do problema da secção crítica? Justifique.
Dica Os três critérios normalmente aceites para uma solução deste problema são: 1) exclusão mútua: não mais do que um processo pode estar dentro da secção crítica; 2) progresso: um processo só poderá esperar se a secção crítica estiver ocupada ou outros processos também pretenderem entrar; 3) ausência de míngua (*starvation*): nenhum processo deve ser impedido de entrar na sua secção crítica indefinidamente.

5- [1.5v] Sobre algoritmos de substituição de páginas em sistemas de memória virtual.

Considere uma arquitetura de computador com um espaço de endereçamento de 24 bits, na qual cada página tem 2^{20} bytes. Assuma, que um computador com esta arquitetura tem 2^{22} bytes de memória física.

Considere a seguinte sequência de acessos onde cada valor nesta sequência representa o nº da página virtual acedida: 6, 15, 4, 10, 2, 6, 10, 4, 15, 4.

Assuma que as *frames* estão inicialmente vazias. Para cada um dos seguintes algoritmos de substituição de páginas:

1. FIFO
2. *Least Recently Used (LRU)*
3. Óptimo (OPT)

indique o nº de *page faults* e o conteúdo de cada uma das *frames* ao longo desta sequência de acesso.

Para cada algoritmo, justifique a substituição da primeira página.

6- [4v] Pretende-se desenvolver uma aplicação de teste da comunicação bidirecional entre 2 processos usando *pipes* anónimos. A ideia é criar um programa em que o processo filho ecoa através dum *pipe* as *strings* que recebe do processo pai por outro pipe. (Lembre-se que os *pipes* são canais de comunicação unidirecionais.) Para transferir dados através dos *pipes* o seu programa deverá usar apenas chamadas ao sistema.

O fragmento de código seguinte ilustra uma função que é executada pelo processo processo filho imediatamente após ter sido criado via *fork()* e ter fechado as extremidades dos *pipes* que não usa:

```
// in: reading end of the pipe used to receive data
// out: writing end of the pipe used to send data
static void child(int in, int out) {
    char buf[BUF_SIZE];
    int n, p;
    struct pollfd fds = { .fd = in, .events = POLLIN };

    while( (p = poll(&fds, 1, 1000)) == 1 ) {
        if (fds.events == POLL_IN ) {
            if( (n = read(in, buf, BUF_SIZE)) == - 1) {
                perror("child: read");
                continue;
            }
            if( n == 0 ) // no more data to receive
                break;

            buf[n] = 0;
            if( write_str(out, buf) == - 1 ) {
                perror("child: write");
                continue;
            }
        }
    }
    close_c(in);
    close_c(out);
    exit( p == -1);
}
```

6.1- Implemente a função `int write_str(int fd, char *str)` que garantidamente escreve completamente a *string* `str`, incluindo o carácter *end of string*, no ficheiro cujo descritor lhe é passado como primeiro argumento, excepto em caso de erro. O valor de retorno de `write_str()` deverá ser 0 em caso de sucesso e -1 em caso de erro.

6.2- Implemente a parte restante da aplicação. O processo pai deverá enviar para o filho os argumentos da linha de comando, excepto o nome do programa. Para o efeito poderá invocar a função `write_str()` definida na alínea anterior. O processo pai deverá ainda imprimir na saída por omissão (*standard out*) os dados que recebe do filho.

Não se esqueça de:

- criar os pipes a usar na comunicação;
- fechar as extremidades dos pipes que não são usadas por cada um dos processos (assuma que as constantes simbólicas `PIPE_IN` e `PIPE_OUT` foram definidas e especificam qual a extremidade a usar para ler e para escrever, respetivamente);
- fechar a extremidade de envio, após ter enviado todas as *strings* para o processo filho

Segue-se um fragmento da *man page* da chamada ao sistema `pipe`, bem como os protótipos de chamadas ao sistema que lhe poderão ser úteis. Não se preocupe com os ficheiros de inclusão.

```
PIPE(2)                          Linux Programmer's Manual                          PIPE(2)

NAME
       pipe, pipe2 - create pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);

DESCRIPTION
       pipe()  creates  a pipe, a unidirectional data channel that can be used for interprocess communication.
       The array pipefd is used to return two file descriptors referring to the ends of the  pipe.   pipefd[0]
       refers  to  the  read end of the pipe.  pipefd[1] refers to the write end of the pipe.  Data written to
       the write end of the pipe is buffered by the kernel until it is read from the read  end  of  the  pipe.
       For further details, see pipe(7).

       If  flags  is 0, then pipe2() is the same as pipe().  The following values can be bitwise ORed in flags
       to obtain different behavior:

       O_NONBLOCK  Set the O_NONBLOCK file status flag on the two new open file descriptions.  Using this flag
                   saves extra calls to fcntl(2) to achieve the same result.

       O_CLOEXEC   Set  the close-on-exec (FD_CLOEXEC) flag on the two new file descriptors.  See the descrip-
                   tion of the same flag in open(2) for reasons why this may be useful.

RETURN VALUE
       On success, zero is returned.  On error, -1 is returned, and errno is set appropriately.

ERRORS
       EFAULT pipefd is not valid.

       EINVAL (pipe2()) Invalid value in flags.

       EMFILE Too many file descriptors are in use by the process.

       ENFILE The system limit on the total number of open files has been reached.
 Manual page pipe(2) line 1
```

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 e Pipes

```
int pipe(int pipefd[2]);
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
```