

# Fundamentos de Sistemas de Operação MIEI 2018/2019

2º Teste, 11 de dezembro 2018, 2 horas

Nº \_\_\_\_\_ Nome \_\_\_\_\_

**Avisos:** Sem consulta; a interpretação do enunciado é da responsabilidade do aluno; se necessário indique a sua interpretação. No fim deste enunciado encontra os protótipos de funções que lhe podem ser úteis.

## Questão 1 (1.5 valores)

Considere um sistema de ficheiros baseado nos princípios do UNIX/LINUX e a operação `mount(nome_do_disco, nome_diretoria)`. Explique porque é que esta operação é necessária, e que acções são feitas pelo sistema operativo quando ela é invocada.

## Questão 2 (2,5 valores)

Para um sistema de ficheiros UNIX/LINUX indique as leituras e escritas que são feitas no disco, quer na zona de dados quer na zona de meta-dados, quando no *shell* (interpretador de comandos) se dá o comando para apagar um ficheiro:

`rm /tmp/XX`

Suponha que

- o utilizador que dá o comando tem permissões para ler e escrever em todas as diretórias envolvidas
- o contador de referência no i-node de `/tmp/XX` está a 1.

**Questão 3 (1,5 valores)** Considere o i-node de um sistema UNIX em que em cada i-node estão 15 endereços de blocos. Há 13 endereços diretos e 2 blocos com endereços de blocos, que por sua vez contêm endereços (endereçamento indireto simples). Sabendo que cada bloco tem 2048 bytes e cada endereço de bloco ocupa 8 bytes, calcule o tamanho máximo que um ficheiro pode ter.

**Questão 4 (2,0 valores)** Os programas de verificação de consistência do sistema de ficheiros, como o *fsck* do UNIX/LINUX fazem várias verificações, nomeadamente envolvendo o conteúdo do mapa de ocupação de blocos com o conteúdo da tabela de *i-nodes*. Explique qual a verificação que é feita e diga como são resolvidas as inconsistências encontradas.

**Questão 5 (2,5 valores)** Considere que num sistema informático com discos de grande dimensão houve um *crash* por motivo desconhecido e o sistema está a arrancar de novo; uma das fases do arranque é verificar a consistência dos discos que vão ser montados. Os sistemas de ficheiros contidos nos discos têm *journal*. Explique como é que, neste caso, vai ser garantido que o sistema de ficheiros está consistente e quais são as vantagens em relação a um programa de verificação de consistência tradicional (isto é, sem *journal*).

**Questão 6 (3 valores)** Pretende-se usar uma máquina com múltiplos processadores para verificar quantos números gerados aleatoriamente e guardados num vetor são primos. O código a usar é o seguinte em que NPROCS representa o número de threads a usar. Suponha que SIZE é múltiplo de NPROCS.

```
#include <pthread.h>
#define NPROCS 4
#define SIZE (10*1024*1024)
int *array;
int count = 0;

int is_prime(int n) {
    for (int i = 2; i <= sqrt(n); i++)
        if (n % i == 0)
            return 0;
    return 1;
}
void *func(void *arg) { // preencher o corpo da função executada por cada thread

}

int main( int argc, char *argv[]){
    pthread_t tids[NPROCS];

    array= (int *)malloc(SIZE*sizeof(int));
    srand(0);
    for (int i=0; i < SIZE; i++) { array[i] = rand() % 5000; }

    for( int i=0; i < NPROCS; i++)
        pthread_create( &tids[i], NULL, func , _____ );

    for( int i=0; i < NPROCS; i++)
        pthread_join( tids[i], _____ );

    printf("Numero de numeros primos no vetor = %d\n", count);
    return 0;
}
```

Complete o código acima. Serão valorizadas soluções que minimizem o número de operações de sincronização realizadas.

## Questão 7 (2,5 valores)

Considere a biblioteca *mySocketTCP* para uso de sockets TCP com as operações seguintes

Operação	Parâmetros de entrada	Retorno
<code>s = serverSocket(port)</code>	port é a porta TCP em que são aceites ligações	s é o canal de entrada / saída associado ao socket criado
<code>sc = acceptServerSocket(ss)</code>	ss é o canal retornado pela função serverSocket	sc é o canal de entrada saída que permite dialogar com o cliente
<code>s = connectSocket(máquina, p)</code>	Máquina é o nome simbólico da máquina onde está o servidor; p é a porta onde o servidor aguarda ligações	S é o canal usado para dialogar com o servidor
<code>nw = writeSocket(s, b, n)</code>	s é o canal a usar; b é o endereço inicial da sequência de bytes a escrever; n o nº de bytes a escrever	nw é o nº de bytes efetivamente escrito
<code>nr = readSocket(s, b, n)</code>	s é o canal a usar; b é o endereço inicial do buffer de bytes onde se recebe; n o nº máximo de bytes a receber	nr é o nº de bytes efetivamente lido
<code>closeSocket(s)</code>	s é o canal usado na ligação	Não tem

Pretende-se escrever o código de um servidor de eco concorrente usando a biblioteca anterior e a API dos Pthreads. Um servidor de eco tem um ciclo eterno em que recebe ligações dos clientes, lê uma sequência de bytes do socket e envia esses bytes de volta. Um cliente de eco teria o seguinte código

```
#include "mySocketTPC.h"
char msg[10] = "123456789";
char buf[11];
int main(){
    int s = connectSocket("www.di.fct.unl.pt", 12345);
    writeSocket(s, msg, 10);
    int n = readSocket(s, buf, 10);
    closeSocket(s);
    buf[n] = '\n'; write(1, buf, n+1);
    exit(0);
}
```

Complete o código do servidor que corre na máquina `www.di.fct.unl.pt` e que está atento à porta 12345:

```
#include <pthread.h>
#include "mySocketTPC.h"

void *func(void *arg) { // preencher o corpo da função executada por cada thread
}

int main( int argc, char *argv[] ){
    int pthread_t tid;
```

```

int s = serverSocket( 12345 );
while( 1 ){
    int sc = acceptServerSocket( s );

    pthread_create( &tid, NULL, func, _____ );
}
// resto do código do servidor. Não se pretende que escreva nada aqui
return 0;
}

```

## Questão 8 (2.5 valores)

Recorde o TPC2; usando a API dos Pthreads pretende-se implementar o mecanismo de sincronização *barreira*. Sobre uma barreira estão definidas duas operações

- **init\_barrier ( B, nProc )** que cria a barreira B e define que esta vai ser usada por *nProc* threads
- **barrier ( B )** bloqueia o thread invocador até que *nProc* threads tenham chamado esta operação.

Uma barreira será definida da seguinte forma:

```

struct barrier {
    pthread_mutex_t mutex; // mutex para acesso exclusivo ao estado da barreira
    pthread_cond_t cond; // Condição em que se bloqueiam os threads à espera da
    // chegada
    int number_threads; // Número de threads que usam a barreira para
    sincronização
    int threads_waiting; // Número de threads que já chamaram barrier.
};

typedef struct barrier barrier_t;

```

Implemente as duas seguintes funções:

```

void initBarrier(barrier_t* bar, int nProcs){
    // Inicializa a estrutura de dados bar, preparando-a para ser usada por nProcs threads
}

void barrier(barrier_t* bar){
    // Bloqueia-se se ainda nem todos os nProcs chamaram a função. Caso contrário,
    // prossegue // e acorda os restantes nProc - 1 threads
}

```

**Questão 9 (2 valores)** Dado um sistema de ficheiros com uma única directória, guardada num único bloco em disco, complete a implementação da função `fs_size` que calcula o espaço (em bytes) ocupado por todos os ficheiros no sistema de ficheiros. Cada ficheiro é representado por uma instância da estrutura `fs_dirent` em que o campo `st` contém o valor `FILE` (valores diferentes de `FILE` indicam que a entrada na directória não está em uso).

```

#define BLOCKSZ      1024      // block size
#define FNAMESZ      11        // file name size
#define FBLOCKS       8         // 8 block indexes in each dirent
#define DIRENTS_PER_BLOCK (BLOCKSZ/sizeof(struct fs_dirent))

struct fs_sblock {      // the super block
    uint16_t magic;      // the magic number
    uint16_t fssize;     // total number of blocks (including the superblock)
    uint16_t dir;        // the number of the block storing the directory
};

struct fs_dirent { // a directory entry (dirent/extent)
    uint8_t st;          // st = FILE if the dirent contains a file
    char name[FNAMESZ]; // the name of the file
    uint16_t size;       // the size of the file
    uint16_t blocks[FBLOCKS]; // disk blocks with file content (zero value = empty)
};

union fs_block { // generic fs block. Can be seen with all these formats
    struct fs_sblock super;
    struct fs_dirent dirent[DIRENTS_PER_BLOCK];
    char data[BLOCKSZ];
};

int fs_size() {
    if (superB.magic != FS_MAGIC)
        return -1; // not mounted

    disk_read(superB.dir, _____); // reads the block storing the directory
                                  // from disk to memory
}

}

```

ANEXO - funções úteis

```
int open( char *fname, int flags,... /*int mode*/ )
int close( int fd )
int read( int fd, void *buff, int size )
int write( int fd, void *buff, int size )
int pipe( int fd[2] )
int dup( int fd )
int dup2( int fd, int fd2 )
pid_t fork(void)
int execve( char *exfile, char *argv[], char*envp[] )
int execvp( char *exfile, char *argv[])
int execlp( char *exfile, char *arg0, ... /*NULL*/
int wait( int *stat )
int waitpid( pid_t pid, int *stat, int opt )
void* memcpy(void* dst, const void* src, size_t n);

int pthread_create( pthread_t *tid, pthread_attr_t *attr,
                    void *(*function)( void* ), void *arg )
int pthread_join( pthread_t tid, void **ret )
int pthread_mutex_init( pthread_mutex_t *mut, pthread_mutexattr_t *attr )
    ou mut = PTHREAD_MUTEX_INITIALIZER
int pthread_mutex_lock( pthread_mutex_t *mut )
int pthread_mutex_unlock( pthread_mutex_t *mut )
int pthread_cond_init( pthread_cond_t *vcond, pthread_condattr_t *attr )
    ou vcond = PTHREAD_COND_INITIALIZER
int pthread_cond_wait( pthread_cond_t *vcond, pthread_mutex_t *mut )
int pthread_cond_signal( pthread_cond_t *vcond )
int pthread_cond_broadcast( pthread_cond_t *vcond )
```