# Fundamentos de Sistemas de Operação MIEI 2018/2019

## 1º Teste, 08 Outubro 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 operação que suporta múltiplos processos, tendo o suporte hardware habitual (interrupções, instruções máquina que geram interrupções por software, CPU com dois modos, MMU, …). Num programa utilizador, explique porque é que o tempo gasto na chamada de uma função definida no próprio programa é menor do que o consumido numa chamada ao sistema como *getpid( )* ou *getuid( )*.

## Questão 2 (2,0 valores)

Descreva o conteúdo do descritor de um processo (*process descriptor* ou *process control block*).

## Questão 3 (1,5 valores)

Considere um sistema onde o escalonador de CPU usa uma única fila de processos prontos (READY queue) e um *time slice T*. Um processo que consome mais tempo de CPU do que T, vai perder o CPU pelo menos uma vez e ser mantido no estado READY até que o CPU lhe volte a ser atribuído. Explique de que forma é que o sistema operativo preserva o estado da computação que o processo está a fazer, de forma a que os resultados obtidos pelo programa sejam os mesmos que seriam conseguidos se o processo ocupasse sozinho o CPU.

## Questão 4 (2,0 valores)

Suponha que um determinado sistema de operação tem um escalonador com duas filas de processos prontos, QA e QB, sendo que QA tem maior prioridade do que QB. Neste caso, os processos em QB só executam quando QA está vazia. Quando um processo é criado é colocado na fila QA. Para cada fila usa-se um algoritmo *Round Robin* e um *time slice* T.

a) Considere que se um processo da fila QA usar todo o seu time slice T, é recolocado na fila QB. Explique porque é que esta abordagem favorece os processos *I/O bound*.

b) A abordagem descrita em a) não está conforme com os objetivos que um escalonador de processos deve ter. Diga porquê e que alterações deveriam ser feitas ao algoritmo no sentido de melhorar a sua funcionalidade.

## Questão 5 (2,0 valores)

Considere um sistema operativo que gere a memória usando páginas e paginação a pedido. Suponha que se fazem as seguintes chamadas ao sistema (em que supõe que o ficheiro existe e tem as proteções adequadas).

```
f = open( "xpto", O_RDWR);
unsigned char *p = (unsigned char *) mmap( …, f, …, PROT_READ | PROT_WRITE, …)
```

a) Indique duas formas distintas de escrever 0x55 no byte com deslocamento 100 do ficheiro.

b) A operação *mmap* atrás referida que alterações irá provocar na tabela de páginas do processo que faz a chamada?

## Questão 6 (1,5 valores)

Considere o algoritmo de substituição de páginas *Clock / 2nd Chance*. O contexto de utilização é de um sistema com múltiplos processos cada um com espaços virtuais de endereçamento com milhares de páginas virtuais. O algoritmo *Clock / 2nd Chance* é invocado sempre que o número de páginas físicas livres desce abaixo de um dado valor e vai examinar os bits de referência de todas as páginas virtuais que estão carregadas em páginas físicas. Explique porque é que são apenas escolhidas, para serem vítimas de substituição, páginas que não são referenciadas há algum tempo.

## Questão 7 (1,5 valores)

Num sistema de gestão da memória física baseado em paginação a pedido é necessário ter uma partição de *swap* (ou um ficheiro que a simule). Explique porque é que este espaço em disco tem de existir.

## Questão 8 (2,0 valores)

Considere um sistema de ficheiros como o UNIX / LINUX e que é feita a chamada ao sistema

```
f = open( "/dir1/f1" , O_RDONLY)
```

Supondo que a diretoria *dir1* existe e que o processo que faz a chamada tem acesso à diretoria raiz e à diretoria *dir1* e permissão de leitura do ficheiro *f1*, detalhe que ações são feitas sobre as seguintes estruturas de dados do sistema:

tabela de canais abertos do processo invocador

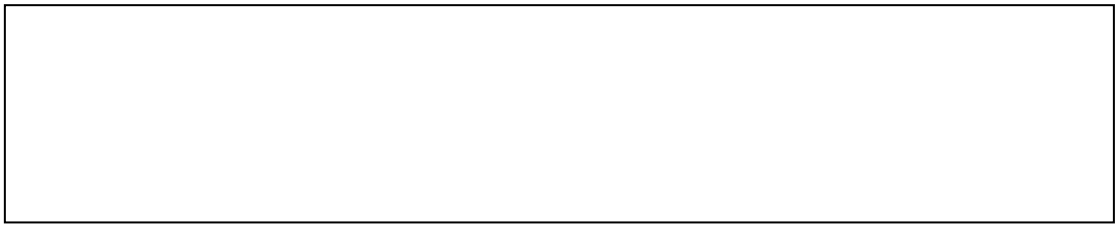
tabela global de ficheiros abertos

tabela de i-nodes (no disco)

## Questão 9 (2.0 valores)

No contexto do TPC1, relembre-se dos valores para o quantum (time-slice) e quota de cada processo, do tipo das filas de processos prontos e do método que lida com o evento "quantum expired". Complete a implementação desse mesmo método preenchendo a caixa no código abaixo.

```
private static final int QUANTUM = 10;
private static final int QUOTA = 20;
private final Queue<ProcessControlBlock<SchedulingState>>[] readyQueues;
public synchronized void quantumExpired(ProcessControlBlock<SchedulingState> pcb) {
    SchedulingState sstate = pcb.getSchedulingState();
    if (sstate.getLevel() == 1) {
            …
     }
     else {
       quota =  sstate.getQuota() - QUANTUM;
       if (quota == 0) {
          Logger.info("Process " + pcb.pid + ": quota expired");
         // actualiza o nível e a quota do processo e coloca-o na fila correspondente
```



```
       }
       …
    }
  }
```

**Questão 10 (2.0 valores)** sobre fork / exec / wait

Considere três ficheiros executáveis *prog1*, *prog2* e *prog3* que estão guardados na diretoria corrente e que são invocados sem argumentos de linha de comando. Pretende-se construir um programa, usando as chamadas ao sistema do UNIX/Linux que efectue a seguinte sequência de ações

- lança a execução de *prog1* e *prog2* em simultâneo

- aguarda que **ambos** terminem

- lança a execução de *prog3*

- aguarda a terminação de *prog3* após o que faz *exit( )*

## Questão 11 (2.0 valores)

Considere uma variante à função *myMalloc( )* implementado nas aulas práticas, em que os metadados são definidos pela seguinte estrutura:

```
typedef struct s_block* t_block;

struct s_block {
  size_t size;  // tamanho do bloco corrente
  size_t used;  // número de bytes do bloco corrente que estão a ser utilizados
  int free;     // flag indicando que o bloco está livre (1) ou ocupado (0)
  t_block next; // apontador para o próximo bloco
};

#define BLOCK_SIZE sizeof(struct s_block)

t_block base; // apontador para o primeiro bloco
```

Note que os campos size e used podem ter valores diferentes. Por exemplo, caso um bloco livre de tamanho 1000 bytes seja utilizado para servir um pedido de alocação de 20 bytes, o conteúdo dos metadados desse bloco será:

```
Size = 1000
Used = 20
Free = 0
Next = endereço do próximo bloco ou NULL, caso este seja o último
```

Implemente a função myRealloc que altera o número de bytes alocados para um dado endereço. Caso o bloco associado ao endereço em causa tenha tamanho suficiente para acomodar o novo pedido, utilize os bytes disponíveis nesse mesmo bloco. Caso contrário procure ou crie um novo bloco: para tal assuma a existência da função *find_or_create_block* com assinatura:

```
t_block find_or_create_block(size_t bytes)
```

A função deve retornar um apontador para o primeiro byte da zona de memória alocada, seja este valor igual ao dado como argumento em ptr ou não.

```
void* myRealloc(void* ptr, size_t bytes) {









}
```

**ANEXO – Chamadas ao sistema**

```
int open( char *fname, int flags,... /*int mode*/ )
int creat( char *fname, int mode )
int close( int fd )
int read( int fd, void *buff, int size )
int write( int fd, void *buff, int size )
int lseek(int fildes, int offset, int whence)
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 *sbrk(unsigned int increment)
void *mmap(void *addr, int len, int prot, int flags, int fd, int offset)
```