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( )*.

A chamada a uma função implica preparar os parâmetros e usar uma instrução máquina como CALL; no final da função executa-se a isntrução máquina RETURN.

Numa chamada ao sistema, executa-se uma instrução máquina INT XX que salva o conteúdo do PC/IP e do SP na pilha, muda o modo do CPU para sistema e que provoca a entrada no código do SO. Aí há verificações dos parâmetros e provavelmente uma mudança de stack. Na saída é executada uma instrução máquina RETURN_FROM_INTERRUPT, que volta a muda o CPU para modo utilizador e faz voltar a execução à instrução a seguir ao INT XX.

O que foi descrito acima leva muito mais tempo do que um CALL / RETURN

Questão 2 (1,5 valores)

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

Um descritor do processo guarda informação sobre cada processo existente no sistema. Essa informação pode ser dividida em duas partes

- relacionada com o estado da computação: espaço para guardar os registos do CPU, a programação da MMU quando o processo está a correr e a tabela de canais abertos

- informação de gestão do CPU, pid, estado do processo, apontadores para filas em que o processo estará inserido, elementos para o algortimo de escalonamento tomar decisões (ex. tempo de CPU consumido recentemente)

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.

Quando um processo perde o processador, o sistema operativo garante que o estado da computação que ele estava a fazer é preservado e que é restaurado quando ele volta a ganhar o CPU. Mais concretamente:

- o estado do CPU virtual (registos, IP, flags) é guardado no descritor do processo e voltado a colocar no CPU real quando o processo volta a ter CPU

- o conteúdo do código, dados, pilha e heap são mantidos na RAM, mas o SO programa a MMU de forma a garantir que nenhum processo consegue ver / alterar o conteúdo de zonas de memória reservadas para outros processos

- as transferências de entrada / sáida em curso são geridas pelo SO, através da informação contida na tabela de canais abertos; essa tabela está guardado na área reservada ao sistema,

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*.

Os processos I/O bound fazem muitas operações de entrada / saída e nunca consomem a sua fatia de tempo até ao fim. Bloqueiam-se para esperar o fim de transferências e portanto mantêm-se na fila QA. Os processos CPU bound vão para fila QB.

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.

Os processos CPU bound podem ficar sem ter CPU muito tempo. Inclusivamente se houver sempre um processo pronto na fila QA, sofrem de *starvation*. Assim sendo, não progridem na sua computação e isso não está de acordo com os princípios de justiça que um algoritmo de escalonamento de CPU deve ter.

Uma forma de resolver o problema seria promover periodicamente os processos que estãona fila QB para a fila QA.

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.

Hipótese 1

```
p[ 100 ] = 0x55;
```

Hipótese 2

```
unsigned char c = 0x55;
lseek( f, 100, INICIO_DO_FICHEIRO);
write( f, &c, 1);
```

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

Vai ser preenchido um conjunto de entradas na tabela de páginas do processo. O nº de entradas preenchido é (comprimento do ficheiro) / (tamanho da página) arredondado para o múltiplo do tamanho de página imediatamente superior. A 1ª página é p / (tamanho da página)

Cada uma das entradas da tabela de páginas vai conter

- bit de validade a 0; bits que permitem a leitura e escrita a 1

- no campo que tem o número da página física, estará o bloco do disco no sisteam de ficheiros que onctém a parte do ficheiro em causa

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.

Sempre que necessita de libertar páginas o algoritmo LRU considera a entrada E na tabela de páginas imediatamente a seguir à última que escolheu como vítima. Se essa página tem a o bit de referência a 1 não é escolhida como vítima, o seu bit de referência é colocado a 0 e passa à página seguinte. Quando chega à última página, volta à primeira. Isto significa que ele só volta a considerar a entrada E após dar uma volta completa à tabela de páginas. Supondo que o espaço de endereçamento é grande, o algoritmo vai demorar algum tempo a dar a volta. Se quando ele considerar novamente a página, o bit de referência ainda está a 0, isso quer dizer que a página já não é refernciada há algum tempo.

Questão 7 (2,0 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.

Num sistema de paginação a pedido, o algoritmo de substituição de páginas escolha como vítima uma página física PF que é a página PV do processo A. Se o processo A volta a precisar da página virtual PV, essa página tem de ser trazida para a memória. Se se tratar de uma página de código, o conteúdo da página PV pode ser obtida no sistema de ficheiros. Se se trata de uma página de dados, do stack ou do heap essa página é obtida na partição de swap.

Quando o processo A perdeu a página PF, o seu conteúdo foi salvaguardado na partição de swap, para que possa ser recuperado se for necessário,

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

A entrada f da tabela de canais abertos é reservada e é preenchida com uma referência para a entrada E da tabela geral de ficheiros abertos

tabela global de ficheiros abertos

A entrada E da tabela de ficheiro abertos é reservada e é preenchida com informação sobre o ficheiro "/dir/f1" nomeadamente qual é o i-node I que lhe corresponde. O campo deslocamento (offset) também é inicializado a 0

tabela de i-nodes (no disco)

O i-node I é trazido para RAM, mas para já nada é feito no disco. Posteriormente, haverá uma alteração da data do último acesso ao ficheiro.

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
```

```
sstate.setLevel(1);

sstate.setQuota(QUOTA);

readyQueues[1].add(pdb);
```

```
       }
       …
    }
  }
```

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( )*

```
int launch(char* prog) {

	switch (fork()) {

		case -1: perror("");
			return 1;

		case 0:
			execlp(prog, prog, NULL);
			perror(prog);
			return 1;
	}

	return 0;
}

int main() {

	if (launch("./prog1"))
		return 1;

	if (launch("./prog2"))
		return 1;

	wait(NULL);
	wait(NULL);

	if (launch("./prog3"))
		return 1;

	wait(NULL);
	return 0;
}
```

Questão 11 (2.0 valores)

Considere uma variante de heap 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) {
  t_block metadata = ((t_block) (ptr – BLOCK_SIZE));

  for (t_block block = base; block != NULL; block++) {
      if (metadata == block) {
            if (metadata.size <= bytes) {
                  metadata.used = bytes;
                  return ptr;
            }
            else break;
      }
   }

   metadata = find_or_create_block(bytes);
   if (metadata == NULL)
      return NULL;

   metadata.used = bytes;
   metadata.free = 0;
   return (metadata+1);
}
```

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)
```