Fundamentos de Sistemas de Operação

FCT UNL 2017/2018

Teste 1, versão A – 30/10/2017

1h45 *Sem consulta. Nas perguntas com múltiplas respostas, assinale a correta com X do lado esquerdo. As respostas erradas descontam 1/4 da cotação da pergunta.*

Q1-(0,6) Os modos supervisor e utilizador do CPU permitem que:

- o SO possa controlar e efetuar a transformação de cada endereço usado pelos processos.
- o SO seja o único software a efetuar operações de escrita na memória.
- os processos possam executar diretamente no CPU sem o SO perder o controlo da arquitetura.
- os processos possam efetuar leituras de ficheiros sem terem de usar a chamada ao sistema *read*.

Q2-(0,6) Um sistema de operação suportando multiprogramação permite rentabilizar os recursos (como o CPU e os periféricos) porque:

- executa sempre vários programas em simultâneo.
- possibilita a existência de programas interativos.
- executa um processo enquanto outro faz IO.
- quando as instruções de um processo vão à memória, pode executar as instruções de outro processo.

Q3-(0,6) A chamada open() permite abrir um ficheiro e criar um canal de acesso. No caso de leitura do ficheiro, como é mantida a posição no ficheiro a partir da qual será efetuada cada nova leitura?

- a chamada ao sistema *read* soma, numa variável local ao processo, o número de bytes lidos, sendo essa variável usada nas próximas leituras.
- a chamada ao sistema *read* mantém, numa variável interna ao SO, o número de bytes lidos, sendo essa variável usada pelo SO nas próximas leituras.
- a chamada ao sistema *read* devolve o número de bytes lidos e o programador usa esse valor para, na chamada seguinte, indicar a posição onde quer ler do ficheiro.
- a chamada ao sistema *read*, soma numa variável associada ao ficheiro, nos metadados, o número de bytes lidos para o SO usar nas próximas leituras desse processo.

Q4-(0,6) Uma chamada ao sistema envolve, tipicamente, as seguintes ações pela ordem indicada:

- coloca nos registos os valores que descrevem o pedido; usa uma instrução especial para chamar o código no SO; o hardware comuta para modo supervisor; o SO executa o código da respectiva chamada.
- usa uma instrução especial para chamar o código no SO; coloca nos registos os valores que descrevem o pedido; o SO executa o código da respectiva chamada.
- coloca nos registos os valores que descrevem o pedido; o hardware comuta para modo supervisor; usa uma instrução especial para chamar o código no SO; o SO executa o código da respectiva chamada.
- usa uma instrução especial para chamar o código no SO; coloca nos registos os valores que descrevem o pedido; o SO executa o código da respectiva chamada; o hardware comuta para modo utilizador.

Q5-(0,6) No contexto da máquina virtual oferecida pelo SO Unix a um processo, a quando da sua criação (fork), numa arquitetura com paginação, a memória virtual desse processo é obtida como?

- Partilhando a tabela de páginas do pai mas atribuindo novas frames (páginas reais) distintas e copiando o conteúdo de todas as frames do pai para o filho.
- Atribuindo uma nova tabela de páginas e novas frames (páginas reais) distintas e copiando o conteúdo de todas as frames do pai para o filho.
- Usando a tabela de páginas do pai e partilhando todas as frames (páginas reais) entre o pai e o filho.
- Atribuindo uma nova tabela de páginas mas partilhando entre o pai e o filho todas as frames (páginas reais) que são apenas lidas por ambos.

Q6-(0,6) Um processo abre um ficheiro para leitura e depois efetua fork(), criando um novo processo que herda esse descritor (canal). Qual o estado desse canal no filho e qual o seu funcionamento?

- Esse descritor (canal) no filho fica fechado pois este é um novo processo.
- Esse descritor (canal) no filho permite que este leia desse mesmo ficheiro, sendo que o pai não irá ler os bytes lidos pelo filho e vice-versa.
- Esse descritor (canal) no filho permite que este leia desse mesmo ficheiro, sendo que o pai vai poder ler todo o conteúdo do ficheiro, independentemente do que o filho lê e vice-versa.
- Esse descritor (canal) pode ser lido pelo filho, mas só depois de o pai escrever algo nesse ficheiro.

Q7-(0,6) Na gestão de memória numa arquitetura com páginas, o SO pode implementar um mecanismo de Copy-on-Write. Com que finalidade?

- Para, sempre que possível, as páginas iguais dos processos serem partilhadas, mas garantindo que cada um tem a sua cópia privada quando necessário, ou seja, após uma primeira escrita.
- Para que, de cada vez que um processo escreve uma página real seja lançada uma interrupção, para que o código do SO copie essa página para uma nova *frame*.
- Para que sempre que um processo escreva uma das suas variáveis, as respectivas páginas sejam copiadas para um novo espaço de endereçamento.
- Para que, sempre que uma página que foi escrita e tem de ser removida da TLB (para dar lugar a outra), esta seja copiada para a tabela de páginas respectiva.

Q8-(0,6) Programas onde o programador usou soluções baseadas em espera ativa degradam a eficiência de todo o sistema multiprogramado, quando vários processos estão prontos a executar (READY)?

- Sim, porque este programa será ineficiente nas esperas ativas quando o CPU for atribuído pelo SO aos outros processos.
- Não, porque este programa corre nos recursos virtuais disponibilizados pelo SO sem interferir com os restantes processos.
- Sim, porque este programa irá usar o CPU, mesmo nas esperas ativas, quando o CPU poderia estar a ser usado pelos restantes processos.
- Não, porque este programa só irá usar o CPU quando este lhe for atribuído pelo SO.

Q9-(0,6) A comunicação entre processos usando um *pipe* caracteriza-se por:

- Permitir a troca de mensagens de forma ordenada entre dois processos sem intervenção do SO.
- Permitir a troca de um stream (fluxo) de bytes entre dois processos sem intervenção do SO.
- Permitir a troca de mensagens de forma ordenada entre dois processos com sincronização pelo SO.
- Permitir a troca de um stream (fluxo) de bytes entre dois processos com sincronização pelo SO.

Q10-(0,6) No contexto de um programa que cria vários processos e threads, estes caracterizam-se por:

- Os threads, ao contrário dos processos, partilham um contexto que inclui o espaço de endereçamento com o programa e dados, os canais de IO e a identificação do utilizador.
- Os processos, ao contrário dos threads, partilham um contexto, incluindo o espaço de endereçamento com o programa e dados, os canais de IO e a identificação do utilizador.
- Os threads, ao contrário dos processos, partilham todo o contexto, incluindo espaço de endereçamento com o programa e dados, os canais de IO e o estado do CPU.
- Os processos, ao contrário dos threads, partilham todo o contexto, incluindo espaço de endereçamento com o programa e dados, os canais de IO e o estado do CPU.

Q11-(0,6) O mecanismo de sinais pode ser usado para notificar os processos de alguns acontecimentos. Indique as situações notificadas com sinais e o tratamento por omissão (*default)* em sistemas do tipo Unix. (nesta pergunta assinale **todas** as que ache verdadeiras)

- o processo acedeu fora do seu espaço de endereçamento válido - sinal ignorado
- o processo fez um acesso a memória que originou uma falta de página - bloqueia o processo
- o processo está bloqueado para ler de um *pipe* e outro processo escreve nesse *pipe* - desbloqueia o processo
- o processo acedeu fora do seu espaço de endereçamento válido - aborta o processo
- o processo fez um acesso a memória que originou uma falta de página - aborta o processo

Q12-(1,4) Complete a figura seguinte com as setas que indicam as transições possíveis entre os estados de um processo e a respectiva ação que provoca a transição de estado, num sistema de operação multiprogramado com *time-sharing* (*time-slices*)

criação —> READY RUNNIG —> *terminação*

BLOCKED

Q13-(2) Um conjunto de três threads (ou processos) partilham as variáveis x e y, inicialmente a zero.

```
p1:                    p2:                    p3:
while ( 1 ){           while ( 1 ) {          while ( 1 ) {
  y = x;                 x = y + 1;             printf("%d %d, ", x,y);
}                      }                      }
```

a) Diga qual/quais as saídas possíveis no *stdout* (assinale **todas** as que ache possíveis)

- 0 0, 0 0, 1 1, 1 1, …
- 1 0, 1 0, 1 1, 1 1, …
- 0 1, 1 1, 1 1, 1 2, …
- 1 0, 1 1, 2 1, 3 1, …
- 10 9, 10 9, 10 10, 10 10, 11 10, ...

b) Copie o código e coloque o controlo de concorrência que ache necessário para que a saída seja sempre: 1 0, 2 1, 3 2, 4 3, ...
(use mutex/condições ou semáforos. Não necessita de os inicializar mas indique os valores iniciais)

Q14-(1) Descreva de que forma um sistema de operação que suporta múltiplos processos concorrentes consegue impedir que um destes fique a executar indefinidamente (por exemplo, a fazer cálculos demorados ou mesmo num ciclo infinito), sem dar a possibilidade dos restantes processos executarem.

Q15-(4,5) Pretende-se implementar um programa de nome "parProc" que lança o processamento paralelo de um ficheiro aplicando-lhe dois comandos distintos, sem argumentos, mas lendo o ficheiro do disco apenas uma vez. Para tal, são criados os processos necessários para executar os comandos, de modo a que cada um receba no seu *standard input* o conteúdo do ficheiro. O processo pai deve ler o ficheiro e enviar o seu conteúdo a ambos os comandos. Cada comando afixado o resultado no *standard output*.
Por exemplo, dado o ficheiro "newsSurf.txt" o resultado desse programa deve ser:

```
$ cat newsSurf.txt
MEO Rip Curl Pro Portugal, Supertubos, 25 de Outubro de 2017 --
Gabriel Medina ganhou pela segunda vez consecutiva, derrotando Julian Wilson na final.
Frederico Morais, Kikas, foi derrotado pelo Brasileiro Filipe Toledo.
$ ./parProc newsSurf.txt wc sort
2   33    220
Frederico Morais, Kikas, foi derrotado pelo Brasileiro Filipe Toledo.
Gabriel Medina ganhou pela segunda vez consecutiva, derrotando Julian
Wilson na final.
MEO Rip Curl Pro Portugal, Supertubos, 25 de Outubro de 2017 --
```

Ou seja, será afixado o que cada um dos comandos "wc" e "sort" escreva. Note que o programa deve terminar só depois de ambos os comandos terminarem. Complete o esqueleto do código que se segue, de acordo com as alíneas seguintes (assuma que não ocorrem erros durante a execução):

a) Implemente a leitura do ficheiro e a escrita do seu conteúdo para cada comando. Cada comando é lançado pela função "launch" que devolve o descritor do canal ligado ao standard input desse processo.

b) Implemente, na função "launch", a criação e a execução de cada comando. Implemente também o código no processo pai que espera que ambos os comandos terminem.

c) Implemente, na função "launch", a criação do meio de comunicação e redirecções que permitam a comunicação (i.e. o envio do ficheiro) entre o processo pai e o standard input do processo filho.

```
. . .
void sys_error( char *msg ) {
      perror(msg);
      exit(1);
}
/* Função que cria e executa o comando definido no parâmetro cmd
   (e.g. "wc", "sort", ou outro) devolvendo o canal ligado ao seu stdin
   através do qual o processo pai vai enviar o conteúdo do ficheiro  */
int launch(char *cmd) {
    pid_t pid;    // identificador do processo filho
```

c)

```
    if ( (pid=fork())== -1 ) sys_error(cmd); // cria novo processo
```

b),c)

```
}
```

```
int main(int argc, char *argv[]) {
   int fdin; // descritor para ler do ficheiro de entrada
   int fd1, fd2;
```

a)

```
   if(argc < 4) {
      printf("Uso: %s ficheiro_entrada comando1 comando2\n", argv[0]);
      return 0;
   }
   // abertura do ficheiro de entrada
   fdin =                                                          a)

   // lançamento dos comandos e obter os canais
   fd1 = launch(                                        );         a)

   fd2 = launch(                                        );

   // Leitura do ficheiro e envio dos seus dados para processamento
   while (                                            ) {         a)









   }
   // O programa tem de esperar a conclusão dos dois comandos
```

b)

```
   close( fdin ); // fecha o ficheiro de entrada
   return 0;
}
```

Q16-(4,5) Determinado programa recorre a múltiplos threads para cada um efetuar determinado cálculo (*worker*). Tem também um thread (*reducer*) para somar todos os cálculos intermédios dos threads produzindo o resultado final.

a) Complete os espaços marcados com a) por forma a criar os threads necessários para executar as funções *worker* e *reducer*, como indicado nos comentários ao longo do código, e também declarar e reservar memória para as variáveis assinaladas.

b) Complete os espaços marcados com b) por forma a garantir que o thread *reducer* só calcula a redução dos resultados computados pelos threads *worker* depois dos últimos terem efetivamente computado o dito resultado. O thread *reducer* não pode utilizar espera ativa.

c) Complete os espaços marcados com c) por forma a retornar ao thread principal (*main*) o resultado da computação efetuada pelo thread *reducer*.

```
// declarações globais
int number_workers;
long *results;
```

```
                                                                    b)
```

```
void* worker(void *arg) {
      long id = (long)(arg);
      results[id] = some_computation(id);
```

```
                                                                    b)
```

```
      return NULL;
}

void* reducer(void *arg) {
```

```
                                                                    b)
```

```
    // Este código só pode executar depois de todos os threads worker
    // terem terminado de executar a função some_computation
    long result = 0;
    for (int i = 0; i < number_workers; i++)
        result += results[i];

    return  _____________________________________;  c)
}
```

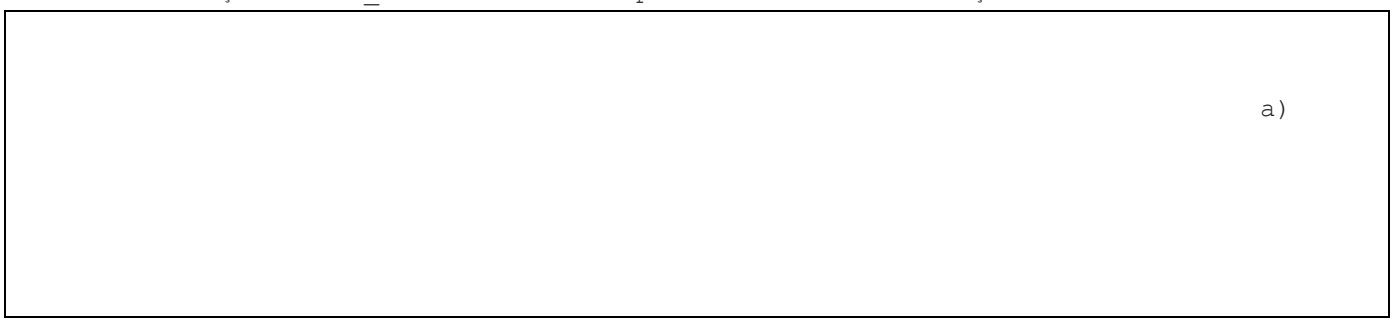
```
int main( int argc, char *argv[] ) {
  // declarações locais
  number_workers = atoi(args[1]);

  results = ____________________________________________________;  a)

  ______________________ worktid[number_workers]; a)

  ______________________ redutid;  a)


  // lança number_workers threads para executarem a função worker
```



```
                                                                  a)
```

```
  // lança um thread para executar a função reducer

  ____________________________________________________________________________ a)


  for (int i = 0; i < number_workers; i++)
     pthread_join(worktid[i], NULL);

  long finalresult;
  pthread_join(redutid, _____________________________________________ );   c)

  printf ("result: %ld\n", finalresult);
  return 0;
}
```

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 )

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 sem_open( char *name, int flags,... /* int mode, int initval */ )
int sem_init( sem_t *sem, int pshare, int initval )
int sem_post( sem_t *sem )  // like V()
int sem_wait( sem_t *sem )  // like P()
```

constantes e flags comuns:

```
NULL
O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_TRUNC
```