Fundamentos de Sistemas de Operação

FCT UNL 2017/2018

Exame, versão A – 15/1/2018

3h00 *Sem consulta. Nas perguntas com múltiplas respostas, assinale a correta com X do lado esquerdo. As respostas erradas descontam 1/5 da cotação da pergunta.*

*Núm:*_____ *Nome:* _____

Q1-(0,6) Os sistemas de operação modernos executam com o CPU em modo supervisor, enquanto garantem que os processos executam com o CPU em modo utilizador. Porquê?

- Porque assim os processos não executam diretamente no CPU, o que facilita a sua gestão.
- Porque assim os processos executam diretamente no CPU com o controlo total sobre todos os periféricos.
- Porque assim os processos não executam diretamente no CPU, tendo que recorrer a chamadas ao sistema para todas as instruções do CPU.
- Porque assim os processos executam diretamente no CPU mas sem poderem aceder aos periféricos, ficando sujeitos às politicas de utilização e partilha impostas pelo SO.

Q2-(0,6) A função de chamada ao sistema write() permite a um processo escrever uma sequencia de bytes num canal que pode representar um ficheiro em disco. Diga o que acontece quando num programa essa função é chamada várias vezes para o mesmo ficheiro.

- De cada vez, a chamada write substitui os bytes no canal dentro do SO, acabando mais tarde por gravar apenas os da última chamada, no respetivo ficheiro em disco.
- De cada vez, a chamada write acrescenta os bytes do processo num buffer interno ao SO, acabando depois este por os gravar todos no respetivo ficheiro em disco.
- De cada vez, a chamada write substitui os bytes no respetivo ficheiro em disco, ficando o ficheiro apenas com a última escrita.
- De cada vez, a chamada write copia os bytes indicados para a posição no canal indicada como argumento do write. A quando do close, o SO vai gravar o estado do canal no respetivo ficheiro em disco.

Q3-(0,6) O nossos programas/linguagens recorrem a chamadas ao sistema. Estas chamadas permitem a um processo pedir operações ao kernel do SO. Compare estas com as chamadas de funções/métodos.

- O mecanismo de chamada é idêntico, executando o código da subrotina chamada, só que na chamada ao sistema os argumentos são passados por registos em vez de usar a pilha.
- O mecanismo de chamada ao sistema usa uma instrução diferente mas, em ambos os casos, o código chamado executa em modo utilizador, para o processo aceder aos dados no SO.
- O mecanismo de chamada ao sistema usa uma instrução diferente que garante a troca dos registos do CPU para que o código do SO não possa aceder aos dados do processo.
- O mecanismo de chamada ao sistema usa uma instrução diferente que muda o CPU para executar o código do kernel em modo supervisor, para que só o SO tenha todo o controlo do hardware.

Q4-(0,6) Na gestão de memória numa arquitetura com páginas, o SO pode implementar um mecanismo de *Copy-on-Write* para que as páginas iguais entre processos sejam partilhadas, poupando espaço de memória real. Explique o funcionamento deste mecanismo.

- Cada página tem indicação se pode ser lida, escrita e executada. Quando uma instrução executa uma dessas operações não autorizadas o CPU gera uma interrupção e o SO copia a página.
- Cada página deve ter indicação da *frame* atribuída na tabela de páginas. Quando uma instrução acede a uma página não atribuída o CPU gera uma interrupção e o SO copia a página.
- Cada página tem indicação de validade. Quando uma instrução escreve numa página inválida o CPU gera uma interrupção e o SO copia a página e torna a cópia válida.
- Cada página tem indicação se pode ser ou não escrita. Quando uma instrução tenta escrever e não pode, o CPU gera uma interrupção e o SO, se for caso disso, copia a página e dá permissão de escrita.

Q5-(0,6) Um SO que suporte múltiplos processos (multiprogramação) permite executar vários programas em simultâneo, partilhando os recursos. Um processo pode assim prejudicar os restantes. Indique das afirmações seguintes uma verdadeira, onde isso acontece:

- Um programa tenta ler de um ficheiro vazio e fica a ocupar o disco e CPU até poder ler.
- Um programa bloqueia-se num lock ou semáforo à espera da sua vez, ocupando memória e CPU de que os outros processos necessitam.
- Um processo bloqueia-se num ciclo de espera a testar uma flag para saber se pode continuar, ocupando o CPU de que os outros processos necessitam.
- Um programa bloqueia-se a ler de um pipe vazio e fica a ocupar CPU de que os outros processos necessitam.

Q6-(0,6) Num sistema suportando processos e threads, estes distinguem-se por:

- Os threads de um processo partilham a memória com o programa e dados, os canais de IO e a identificação do utilizador, enquanto os processos não.
- Os processos partilham a memória com o programa e dados, os canais de IO e a identificação do utilizador enquanto os seus threads não.
- Os threads de um processo partilham a memória com o programa e dados, os canais de IO e o estado do CPU, enquanto os processos não.
- Os processos partilham a memória com o programa e dados, os canais de IO e o estado do CPU enquanto os seus threads não.

Q7-(0,6) Numa chamada *read* quais as ações efetuadas pelo SO?

- Verifica os blocos ocupados no disco para devolver estes ao processo.
- Lê sempre um novo bloco do ficheiro no disco que devolve ao processo.
- Lê, se necessário, um ou mais blocos do ficheiro no disco para um buffer e devolve os bytes pedidos pelo processo.
- Lê o inode indicado pelo nome do ficheiro para obter depois um novo bloco do ficheiro no disco que devolve ao processo.

Q8-(0,6) O escalonamento de processos efetuado pelos SOs multiprogramados com *time-sharing* (*time-slices*) consiste normalmente no quê?

- Garantir que todos os processos têm acesso aos discos sem interferirem entre sí.
- Trocar o CPU entre todos os processos existentes no sistema para os executar rapidamente.
- Dividir o tempo do CPU entre todos os processos que estão prontos a executar.
- Executar um processo depois de outro terminar.

Q9-(0,6) Diga como um sistema do tipo Unix, ao executar um novo programa (chamada execve), pode melhorar o uso de memória e acelerar o inicio da sua execução em hardware com paginação?

- Carregando em páginas contíguas todo o novo programa usando assim menos leituras do disco.
- Carregando para memória apenas o código e adiando a criação das páginas para os dados.
- Usando paginação a pedido para carregar as páginas apenas à medida que vão sendo necessárias para o programa executar.
- Usando paginação e *swap* de páginas para disco para ter mais memória disponível.

Q10-(0,6) Escolha a afirmação correta:

- Um processo diz-se CPU bound se o usa pouco tempo de CPU.
- Um processo diz-se CPU bound se a maior parte do tempo de execução for usando o CPU.
- Um processo diz-se IO bound se fizer In e Out de dados sem ser de ficheiros.
- Um processo diz-se IO bound se não usar ficheiros em disco.

Q11-(0,6) Qual a vantagem de um SO dispor de sistema de escalonamento dos pedidos feitos a um disco?

- Permite acelerar os discos ao dedicar mais CPU aos processos com mais pedidos.
- Permite optimizar o tempo de execução dos processos ao distribuir os pedidos feitos pelos discos mais livres.
- Permite optimizar o CPU ao ordenar os processos pelos pedidos de IO feitos, por forma a reduzir o tempo de execução desses pedidos.
- Permite acelerar o seu atendimento ao ordenar os pedidos de IO de forma a reduzir o movimento das cabeças de leitura/escrita do disco.

Q12-(1,5) Considere os estados possíveis de um processo num sistema de operação multiprogramado com *time-sharing* (*time-slices*).
a) Complete a figura seguinte com as setas que indicam as transições possíveis entre os estados.

criação —> READY RUNNIG —> *terminação*

BLOCKED

b) Indique e explique o eventos que provocam cada uma das transições anteriores.

Q13-(1,5) Explique o que entende por um processo ficar em *starvation* a aguardar por CPU e explique como um SO usando um escalonador baseado em MLFQ - Multi-level Feedback Queue evita que tal aconteça.

Q14-(1,5) Aquando da abertura de um ficheiro é necessário percorrer toda a hierarquia de diretorias para identificar o ficheiro pretendido e seus blocos em disco. Descreva como os sistemas de operação evitam ter de estar constantemente a ler blocos do disco devido aos pedidos dos processos para aceder aos mesmos inodes e blocos das directorias, assim como aos mesmos blocos dos ficheiros usados por vários processos.

Q15-(1,5) Num sistema que suporta semáforos como os vistos nas aulas, pretende-se implementar um mecanismo de exclusão mútua semelhante aos mutex dos threads vistos nas aulas.
Não é necessário uma solução sintaticamente correta, pode usar pseudocodigo e pode assumir um número máximo de mutexes. Implemente as funções seguintes:

- Mutex_init(int id) – inicializa um mutex identificado pelo número em id, caso ainda o não tenha sido.
- Mutex_lock(int id) – adquire mutex indicado; bloqueia o thread se o lock pertencer a outro thread até este ser libertado.
- Mutex_unlock(int id) – liberta o mutex indicado; dá a vez a um dos threads bloqueados à espera deste mutex.

Q16-(2) Pretende-se implementar um programa que, dado um número *n* como argumento, gera os primeiros *n* números primos e escreve-os num ficheiro binário (uma sequência de inteiros sem sinal). Assumindo que o executável se chama *primes*, e que o utilizador quer gerar um ficheiro com os 10 primeiros números primos, uma utilização do programa poderá ser:

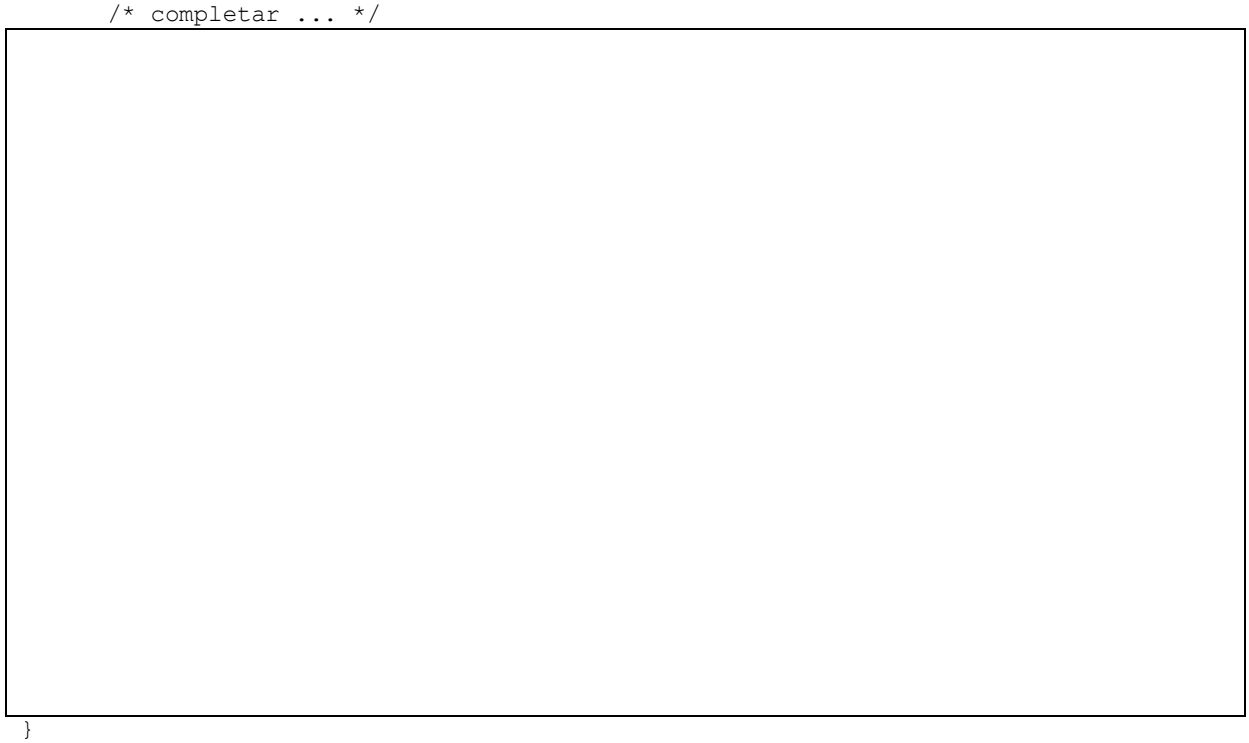
```
$ ./primes fileP 10
$ wc -c fileP
40
$ od -D fileP
2   3   5   7    11   13    17    19   23  29
```

O comando *wc* está a ser usado para mostrar o número de bytes do ficheiro (10*4bytes) e o comando *od* está ser usado para mostrar o conteúdo do ficheiro binário.

a) Pretende-se que utilize as chamadas ao sistema do UNIX/Linux para criar o ficheiro com o número de primos requerido pelo utilizador. Complete o código que se segue, o qual assume a existência de uma função designada "nextPrime" que devolve o próximo primo na sequência. Ou seja, da primeira vez que é invocada, a função nextPrime devolve o valor 2, da segunda vez devolve o valor 3, etc.

```
...
extern unsigned int nextPrime(void);

void writePrimes( int fd, int maxSeq) {
      /* completar ... */
```



```
}

int main(int argc, char *argv[]) {
   if (argc != 3) {
        printf("usage: %s <filename> <number_of_primes>\n", argv[0]);
        exit(1);
   }
   /* create the file for the primes */

   int f =
   writePrimes( f, atoi(argv[2]) );
   return 0;
}
```

b) Assuma agora que, no mesmo SO, pretende usar a facilidade de mapear ficheiros para memória para guardar os números no ficheiro. Complete o código seguinte que, por simplificação, já ajusta a dimensão do ficheiro de modo a poder conter o número de primos pretendido pelo utilizador.

```
...
void fatal_error(char *str) {
  perror(str);
  exit(1);
}

extern unsigned int nextPrime(void);

void writePrimes( int fd, int maxSeq ) {
    ftruncate(fd, maxSeg*sizeof(unsigned int)); // ensure the file has enough space
/* completar ... */
```

```
}

int main(int argc, char *argv[]) {
    /* check for required command line arguments */
    if (argc != 3) {
    printf("usage: %s <filename> <number_of_primes>\n", argv[0]);
    exit(1);
    }
    /* create the file for the primes */

    int f =

    writePrimes( f,  atoi(argv[2]));

    return 0;
}
```

Q17-(1,5) Alguns editores de texto criam ficheiros de *backup*/salvaguarda do ficheiro em edição, com o nome do original acrescido do caracter '~'. Por exemplo, após a edição de dois ficheiros prog.c e myprog.java, o conteúdo duma diretoria é:

```
$ ls
myprog.java myprog.java~ prog.c prog.c~
```

Implemente um programa que, dado o percurso completo ou relativo para uma diretoria, apaga todos os ficheiros de *backup*/salvaguarda que nela possam existir (todos os terminados em '~'). Em caso de omissão da diretoria, o programa assume que se trata da diretoria corrente (i.e. ".").

```
...
void deleteTemp(char * dir);

int main(int argc, char*argv[] ) {
       char *dir = ".";

       if ( argc != 1 ) dir = argv[1];
       deleteTemp(dir);
       return 0;
}

void deleteTemp(char * dir)
{
```

```
}
```

Q18-(2) No contexto da criação e controlo de processos

a) Implemente a função `redirect_stderr` que redirecciona o *standard error* do processo corrente para o ficheiro fornecido no parâmetro `filename`. A função deve retornar 0 caso a operação de redireccionamento tenha sido realizada com sucesso e -1 caso contrário.

```
int redirect_stderr(char* filename)
{









 }
```

b) Implemente a função `run_program` que executa um programa, fornecido através do parâmetro `prog`, com o seu *standard error* redireccionado para o ficheiro `error_file`. O programa a executar não tem argumentos e deve ser executado no contexto de um novo processo, criado para o feito. Além disso, a função deve esperar pela conclusão da execução de `prog`, retornando o valor 0 se a execução foi desencadeada com sucesso e -1 no caso contrário. Para efetuar o redireccionamento deve utilizar a função `redirect_stderr` da alínea anterior, mesmo que não a tenha implementado.

```
int run_program(char* prog, char* error_file)
{









 }
```

c) Que alterações introduziria à solução da alínea anterior para que a função `run_program` retornasse o *exit status* da execução de `prog` (em vez de 0), caso a execução do programa tivesse sido desencadeada com sucesso?

Q19- (1,9) No contexto de controlo de concorrência, considere os threads T1 e T2 a executar os seguintes trechos de código:

```
void *T1( void *arg ) {

for (int i = 0; i < N; i++) {

  printf ("T1 ");

}

}
```

```
void *T2( void *arg ) {

for (int i = 0; i < N; i++) {

   printf ("T2 ");

}

}
```

a) Complete a implementação de ambos os threads para que estes, sincronizadamente, executem as iterações dos seus ciclos de forma alternada, começando por T1. Ou seja, para N = 5, a execução deverá produzir o seguinte resultado:

```
T1 T2 T1 T2 T1 T2 T1 T2 T1 T2
```

b) Declare e inicialize, no retângulo seguinte, todas as variáveis globais que precisar e crie os threads que executam as funções anteriores:

ANEXO

funções e tipos úteis:

```
void *mmap(void *addr, int len, int prot, int flags, int fd, int offset)
int msync(void *addr, int len, int flags)
int munmap(void *addr, int len)
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()
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 stat(char *path, struct stat *buf)
int fstat(int fd, struct stat *buf)
DIR *opendir(const char *filename)
struct dirent *readdir(DIR *dirp)

struct dirent { inot_t d_ino;
                char d_name[NAME_MAX+1];
}

struct stat { dev_t st_dev;
              ino_t st_ino;
              mode_t st_mode;
              int st_nlink;
              uid_t st_uid;
              off_t st_size;
              time_t st_mtime;
          ... };
```

constantes e flags úteis:

```
NULL
O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_TRUNC
PROT_READ, PROT_WRITE,
MAP_SHARED, MAP_PRIVATE, MAP_ANON

S_ISREG(m)    /* regular file */
S_ISDIR(m)    /* directory */
S_ISFIFO(m)   /* named pipe (fifo) */
S_ISLNK(m)    /* symbolic link */
```