LEIC/LETI – 2022/23 – 2 º Exame de Sistemas Operativos

10 Fevereiro de 2023, Duração: 2h00m

- **Responda no enunciado, apenas no espaço fornecido. Identifique todas as folhas.**
- **Por omissão, os excertos de código omitem o tratamento de erros; nas suas respostas com código, pode também omitir o tratamento de erros.**
- Nas perguntas de escolha múltipla **existe apenas uma resposta certa**. Em caso de dúvida, **pode selecionar uma ou duas alíneas**. A nota é calculada pelas alíneas que escolheu na sua resposta, da seguinte forma: a alínea correta conta com a cotação completa; cada alínea incorreta desconta 1/3 da cotação da pergunta.

Grupo 1 [Sistemas de Ficheiros, 3,3]

1) **[0,6v]** Qual das seguintes afirmações acerca do **Virtual File System** (VFS) é **verdadeira**:

a) O objetivo principal do VFS é reduzir a latência de acesso aos sistemas de ficheiros através da utilização de caches dos blocos dos discos.

b) No VFS, o *superblock* guarda a informação sobre os blocos de discos onde são armazenados os inodes dos ficheiros de vários sistemas de ficheiros

c) No VFS, o *superblock* permite determinar quais são as funções que o núcleo deve invocar para aceder aos ficheiros guardados em sistemas de ficheiros heterogéneos.

d) A virtualização do VFS só permite sistema de ficheiros Unix e Linux, não permite por exemplo sistemas baseado em FAT .

2) **[0,6v]** É possível que uma dada entrada da tabela global de ficheiros abertos possa ser referenciada por processos diferentes?

a) Nunca é possível

b) Sim, através da chamada de sistema fork()

c) Sim, através da chamada de sistema dup()

d) Sim, quando dois processos concorrentemente ativos abrem o mesmo ficheiro

3) **[0,6v]** Qual das seguintes afirmações é **verdadeira**:

a) Quando se apaga um *hard link* para um ficheiro, o ficheiro é apagado

b) Quando se apaga um *symbolic link* para um ficheiro, o ficheiro nunca é apagado

c) Quando se apaga um *hard link* para um ficheiro, o ficheiro nunca é apagado

d) Nenhuma das anteriores

4) **[1,5v]** No EXT3, se o tamanho de bloco for 1KB e as referências para os blocos ocuparem 32 bits, quantos blocos de disco são utilizados para armazenar as referências para os blocos de um ficheiro de tamanho 358KB? Na conta, excluir o espaço usado para armazenar o inode do ficheiro.

a) 3

b) . 2

c) 1

d) 0

Grupo 2 [Tarefas e processo, 1.5v]

1) **[1,5v]** Pretende-se implementar um programa que executa sequencialmente, em processos diferentes, todos os programas cujos nomes lhe são passados em input por linha de comando (argc/argv), enquanto não haver um programa cuja execução termina por causa de um signal. Escolha a instrução a usar em cada caixa entre as alíneas indicadas abaixo. Em caso de indecisão pode indicar duas alíneas. Também neste caso cada alínea incorreta desconta 1/3 da cotação.

```
#include <sys/wait.h>
#include <stdio.h>

int main(int argc, char **argv) {
int p, status;
if (argc < 2) return 1;

for (int i=1; i<argc; i++) {
    char *path =  [ d. ]

    p =  [ a. ]
    if (p == 0)      {
       status=[ l. ]
       [ j. ]
  }
       p=[ i. ]
       if [ g. ]
        break;
  }
}
```

a. fork();
b. WIFEXITED(status)
c. wait(NULL)
d. argv[i];
e. argc[i];
f. fork(path);
g. WIFSIGNALED(status)
h. execl(path);
i. wait(&status)
j. return status
k. argv[i+1];
l. execl(path, path, 0);
m. execv(path);
n. WIFSIGNALED(p)
o. fork(status)

Macro: *int* **WIFEXITED** *(int status)* This macro returns a nonzero value if the child process terminated normally with `exit` or `_exit`.
Macro: *int* **WIFSIGNALED** *(int status)* This macro returns a nonzero value if the child process terminated because it received a signal that was not handled.

Grupo 3 [Exclusão mútua, 1,8v]

1. **[0,6v]** Qual das seguintes afirmações sobre as implementações da abstração do mutex é **falsa**:
   a) As instruções que garantem acesso atómico (ex.,Test-and-Set/BTS) à memoria permitem evitar algoritmos complexos em software, como o algoritmo de Lamport
   b) As instruções que garantem acesso atómico à memoria permitem evitar espera ativa caso o mutex seja ocupado
   c) As instruções que garantem acesso atómico à memoria podem ser utilizadas em modo utilizador

Indique d. caso todas as afirmações acima sejam verdadeiras.

____

2. **[0,6v]** Qual das seguintes afirmações é **verdadeira**:
   a. Os mutexes proporcionam menor paralelismo do que os read-write locks
   b. A utilização dos read-write locks é sempre preferível à dos mutexes
   c. A utilização dos mutexes é sempre preferível à dos read-write locks
   d. As afirmações acima são todas falsas.

____

3. **[0,6v]** Qual das seguintes afirmações é **falsa**:
   a) A utilização de try_lock() em vez do que lock() permite evitar o problema da inter-blocagem (*deadlock)*
   b) A utilização de try_lock() em vez do que lock() permite evitar o problema da míngua (*livelocks)*
   c) A utilização ordenada de um conjunto de locks através da chamada lock() permite evitar quer interblocagem quer mingua

Indique d. caso todas as afirmações acima sejam verdadeiras.

____

Grupo 4 [Semáforos e variáveis de condição, 4,4v]

1. **[0,6v]** Qual das seguintes observações é **verdadeira**?
   a) Um semáforo tem sempre de ser utilizado em combinação com um mutex
   b) O contador de um semáforo pode ter valores menores do que 0
   c) Um semáforo binário inicializado com o valor 1 pode implementar um mutex através das funções esperar/assinalar, que correspondem respetivamente a lock/unlock
   d) Um semáforo binário inicializado com o valor 0 pode implementar um mutex através das funções esperar/assinalar, que correspondem respetivamente a lock/unlock

____

2) **[3,8v]** O programa abaixo pretende dar uma solução genérica para o problema de uma tarefa principal esperar pela terminação das tarefas que criou, terminado estas por qualquer ordem.

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

static pthread_cond_t threadDied =  1.                ;
static pthread_mutex_t threadMutex=    2.             ;

static int totThreads = 0;      /* Total number of threads created */
static int numLive = 0;         /* Total number of threads still alive or
                                   terminated but not yet joined */
static int numUnjoined = 0;     /* Number of terminated threads that
                                   have not yet been joined */
enum tstate {                   /* Thread states */
    TS_ALIVE,                   /* Thread is alive */
    TS_TERMINATED,              /* Thread terminated, not yet joined */
    TS_JOINED                   /* Thread terminated, and joined */
};

static struct {                 /* Info about each thread */
    pthread_t tid;              /* ID of this thread */
    enum tstate state;          /* Thread state (TS_* constants above) */
    int sleepTime;              /* Number seconds to live before terminating */
} *thread;

static void *                   /* Start function for thread */
threadFunc(void *arg)
{
    int idx = (int) arg;
    int s;

    sleep(thread[idx].sleepTime);       /* Simulate doing some work */

    s=  3.

    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    numUnjoined++;
    thread[idx].state = TS_TERMINATED;

    s = pthread_mutex_unlock(&threadMutex);
    if (s != 0)
        errExitEN(s,"pthread_mutex_unlock");

    s = pthread_cond_signal(   7.          );
    if (s != 0)
        errExitEN(s, "pthread_cond_signal");

    return NULL;
}

int
main(int argc, char *argv[])
{
    int s, idx=10;

    srand=(time(NULL));
    /* Create all threads */
```

```
    for (idx = 0; idx < argc - 1; idx++) {
        thread[idx].sleepTime = rand()%10;
        thread[idx].state = TS_ALIVE;
        s = pthread_create(&thread[idx].tid, NULL, threadFunc, (void *) idx);
        if (s != 0)
            errExitEN(s, "pthread_create");
    }

    totThreads = argc - 1;
    numLive = totThreads;

    while (numLive > 0) {
        s = pthread_mutex_lock(&threadMutex);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

        4.                  (numUnjoined == 0) {

            s = pthread_cond_wait(&threadDied, &threadMutex);
            if (s != 0)
                errExitEN(s, "pthread_cond_wait");
        }

        for (idx = 0; idx < totThreads; idx++) {
            if (thread[idx].state == TS_TERMINATED) {
                s = pthread_join(thread[idx].tid, NULL);
                if (s != 0)
                    errExitEN(s, "pthread_join");

                thread[idx].state = TS_JOINED;
                numLive--;
                numUnjoined--;

                printf("Reaped thread %d (numLive=%d)\n", idx, numLive);
            }
        }

  s=    6.

        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");
    }

    exit(EXIT_SUCCESS);
}
```

2.i) **[2,0v]** Preencha no programa as instruções em falta

1. PTHREAD_COND_INITIALIZER
2. PTHREAD_MUTEX_INITIALIZER;
3. pthread_mutex_lock(&threadMutex);
4. while
5. if
6. pthread_mutex_unlock(&threadMutex);
7. &threadDied
8. &threadMutex

2.ii) **[0,6v]** Qual das seguintes afirmações é **verdadeira**?

a) O programa usa as *condition variables* para detetar quando uma tarefa termina sem utilizar signals
b) O programa usa as *condition variables* para garantir que as varáveis partilhadas são atualizadas em exclusão mútua
c) O programa usa as *condition variables* para detetar quando uma tarefa termina porque se uma tarefa termina e a tarefa principal não estiver bloqueada em pthread_join() perde-se esta informação
d) O programa usa as *condition variables* porque o pthread_join() obriga a indicar qual a tarefa porque se espera sendo impossível esperar por uma tarefa qualquer

2.iii) **[0,6v]** No programa existe uma sequência de instruções

A
```
  s = pthread_mutex_lock(&threadMutex);
…….
  s = pthread_cond_wait(&threadDied, &threadMutex);
```
funcionamento semelhante poderia ser obtido com semáforos

B
```
  s = pthread_mutex_lock(&threadMutex);
…….
  S = semwait(&threads);
```
Considere a primeira sequência A e segunda sequência B. Qual das seguintes afirmações é **verdadeira**?

a) A pode conduzir a deadlock e B não pode
b) A pode conduzir a deadlock e B também
c) A não pode conduzir a deadlock e B pode
d) A não pode conduzir a deadlock e B também não

2.iv) **[0,6v]** Analise a variável de condição threadDied e a situação em que várias tarefas terminam simultaneamente. Qual das seguintes afirmações é **verdadeira**?

a) A variável de condição não tem memória de eventos e, portanto, o programa falha nesta situação
b) Tem memória dos eventos somando o número de vezes que houve chamadas a pthread_cond_signal()
c) Se usasse semáforos este teria memória do número de vezes que foi assinalado
d) Esta situação nunca ocorre porque usando números aleatórios para o tempo de sleep() as tarefas vão terminar sempre em instantes diferentes

Grupo 5 [Signals, 2,4v]

Considere o programa abaixo

```
static void
sigHandler(int sig){
    static int count = 0;
    if (sig == SIGINT) {
        count++;
        return;  /* Resume execution */
    }
    _exit(EXIT_SUCCESS); //async_signal_safe version of exit()
}
int
main(int argc, char *argv[])
{
    if (signal(SIGINT, sigHandler) == SIG_ERR)
        errExit("signal");
    if (signal(SIGQUIT, sigHandler) == SIG_ERR)
        errExit("signal");
    for (;;) /* Loop forever, waiting for signals */
      [          ]
}
```

1) **[0,6v]** Que instrução colocaria na caixa vazia no programa acima
   a) pause()
   b) wait()
   c) pthread_join()
   d) pthread_cond_wait()

[ ]

2) **[0,6v]** Qual o resultado da execução deste programa supondo a semântica BSD:
   a) Conta o número de SIGINT e termina
   b) Conta o número de SIGINT, não terminando e só termina quando recebe SIGQUIT
   c) Conta o número de SIGINT e de SIGQUIT e não termina
   d) O valor do argumento sig não foi definido na chamada a signal inicial pelo que o if não vai funcionar

[ ]

3) **[1,2v]** Considere o programa abaixo. Suponha que antes do fork() o processo pai ignorou todos os signals.

3.i) **[0,6v]** Considere que o processo pai queria garantir que o processo filho termina, efetue o preenchimento dos parâmetros na caixa A

```
switch (childPid = fork()) {
case -1:
      errExit("fork");
case 0:
      printf("Child (PID=%ld) \n", (long) B );
      pause();
      exit(EXIT_SUCCESS);
default:
      sleep(3);
      if (kill( A ) == -1)
          errMsg("kill");
      sleep(3); /* Give child a chance to react to signal */
      exit(EXIT_SUCCESS);
}
```

a) (childPid, SIGINT)
b) (getpid(), SIGKILL)
c) (childPid, SIGKILL)
d) (childPid, qualquer signal cuja opção por omissão seja terminar o processo uma vez que o processo filho não efetuou qualquer tratamento dos signals)

3.ii) **[0,6v]** Qual a substituição adequada para a caixa B:

a) childPid
b) getpid()
c) getppid ()
d) childPid()

Grupo 6 [Pipes e escalonameto, 3.3v]

1) **[1,2v]** O processo A vai ler de um FIFO já existente, executando a função `read(fdpipe,@buffer,30).`
A tabela seguinte descreve vários estados de utilização do FIFO, escolha a linha correta, considerando que X é o número de bytes existente no FIFO.

X=0, extremidade de escrita aberta	X=0, extremidade de escrita fechada	X<30	X>=30

O preenchimento da tabela é
a)

Bloqueia	retorna EOF (0)	retorna X bytes	retorna 30 bytes

b)

Retorna EOF (0)	retorna EOF (0)	retorna X bytes	retorna 30 bytes

c)

Bloqueia	Bloqueia	retorna X bytes	retorna 30 bytes

d)

Bloqueia	Retorna EOF (0)	Bloqueia até receber 30 bytes	retorna 30 bytes

2)**[1.5v]** No algoritmo de escalonamento Unix apresentado nas aulas teóricas, se um processo utiliza o CPU dura 10 segundos numa fase inicial e a seguir fica indefinidamente bloqueado, a qual valor de prioridade converge no limite depois de um tempo infinito? Ao responder supor PrioridadeBase=10 e nice=0.

a) 10
b) Infinito
c) 0
d) 5

3) **[0.6v**] Assinale qual das seguintes afirmações acerca do escalonamento CFS (Completely Fair Scheduler) é **falsa**.

a) Os processos executáveis são mantidos numa red-black tree ordenada por vruntime.
b) A escolha da min_granularity pode afetar a latência de comutação.
c) O time-slice tem duração fixa.
d) O valor de vruntime determina qual o processo mais prioritário

Grupo 7 [Chamadas de sistemas Linux, 2,1v]

1) **[0,6v]** Quando executa uma *system call,* um processo em Linux pode bloquear-se no núcleo numa *wait queue*. Qual das seguintes afirmações é **verdadeira**?
   a) O processo será novamente tornado executável sempre que um *signal* lhe é enviado
   b) O processo só será desbloqueado quando o evento porque espera é despoletado

c) O processo quando é acordado por um evento necessita de verificar se a condição que levou a acordar ainda é válida. Senão tem de bloquear-se novamente, mesmo quando recebe um signal.

d) O processo quando é acordado por um evento necessita de verificar se a condição que levou a acordar ainda é válida senão tem de bloquear-se. Se se bloqueou com *wait_interruptible* os *signals* desbloqueiam o processo e não verifica se a condição de bloqueio se tornou válida

2) **[1,5v]** Considere a execução do programa seguinte num sistema Linux:

```
1 int main() {
2     int x = 0;

3     if (fork() == 0) {
4         x += 1;
5     }

6     printf("x=%d\n", x);
7     return 0;
8 }
```

Assuma que antes da chamada a `fork` a tabela de páginas do processo contém apenas duas páginas, uma para o programa, outra para a pilha (por simplificação vamos ignorar a existência da região de dados e *heap*):

Válida	Presente	Page Number	Base	Proteção (R: read. W: write, X: execute, C: Copy-on-Write)
Sim	Sim	0x10	0x01	R - X -
Sim	Sim	0x20	0x02	R W - -
Não	-	-	-	-

2.a) [0,75v] Considere o estado do processo **filho** na linha 3 (imediatamente após o retorno da chamada `fork`). **Preencha** a tabela de páginas deste processo.

Caso seja necessário alocar novas páginas **virtuais**, use os números de página `0xC0`, `0xC1`, ou `0xC2`.
Caso seja necessário alocar novas páginas **físicas**, use os números de página `0xD0`, `0xD1`, ou `0xD2`.

Válida	Presente	Page Number	Base	Bits de Proteção
Sim	Sim	0x10	0x01	R - X -
Sim	Sim	0x20	0x02	R - - C

Nota : A resposta acima está de acordo com o que a maioria das arquitecturas fazem utilizando os registos para retorno das chamadas sistema. Neste caso a pilha não é utilizada e a função em c que faz o wrapper da *system call* devolve os valores de retorno do fork correctamente.

Contudo, poder-se-ia numa visão mais simplista considerar que o retorno da função era colocado no pilha o que levaria a que a pilha do processo filho fosse alterada pelo núcleo de forma a colocar o valor de retorno da fork(). Tendo isto em conta foi considerada válida também a seguinte resposta alternativa:

Válida	Presente	Page Number	Base	Bits de Proteção
Sim	Sim	0x10	0x01	R - X -
Sim	Sim	0x20	0xD0	R W - -
Não	-	-	-	-

2.b) [0,75v] Considere agora o estado do processo **filho** na linha 7.
Preencha a tabela de páginas deste processo.

Caso seja necessário alocar novas páginas **virtuais**, use os números de página 0xE0, 0xE1, ou 0xE2.
Caso seja necessário alocar novas páginas **físicas**, use os números de página 0xF0, 0xF1, ou 0xF2.

Válida	Presente	Page Number	Base	Bits de Proteção
Sim	Sim	0x10	0x01	R - X -
Sim	Sim	0x20	0xF0	R W - -
Não	-	-	-	-

À segunda alternativa corresponderia

Válida	Presente	Page Number	Base	Bits de Proteção
Sim	Sim	0x10	0x01	R – X –
Sim	Sim	0x20	0xD0	R W – –
Não	-	–	–	–

a Grupo 8 [Gestão da Memória, 1.2v]

1) **[0.6v]** Qual das seguintes afirmações é **verdadeira**.
   a) Num sistema com paginação não pode haver fragmentação externa.
   b) Num sistema com paginação não pode haver fragmentação interna.
   c) Num sistema com segmentação não pode haver fragmentação externa.
   d) Nenhuma das anteriores

2) [0.6v] Qual das seguintes afirmações é **falsa**:
   a) Cada processo tem uma tabela de páginas distinta
   b) Quando existe uma comutação de processo, o núcleo altera os registos do *Memory Management Unit* para apontar para a tabela de páginas do processo
   c) Sempre que houver um TLB miss, o núcleo é acordado
   d) Sempre que houver um Page fault (página não presente em RAM), o núcleo é acordado.