LEIC/LETI – 2022/23 – 1º Exame de Sistemas Operativos

27 de janeiro 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, 2.7v]

1) [0.6v] Se o directório corrente de um processo é '/home/joao/SO' e o processo executa:
open(fd, './../../luis/ASA/README.txt')

Qual é o nome absoluto do ficheiro que o processo tenta aceder:

a) /home/joao/SO/luis/ASA/README.txt
b) /home/luis/ASA/README.txt
c) /home/joao/luis/ASA/README.txt
d) / luis/ASA/README.txt'

2) [0.6v] Qual é a vantagem principal da utilização da organização baseada em diretórios em comparação à organização em listas para um sistema de ficheiros:

a) Reduz a dimensão dos metadados guardados pelo file-system
b) Permite evitar fragmentação externa
c) Acelera a procura dos ficheiros por nome
d) Nenhuma das anteriores

3) [1.5v] Considere o seguinte programa:

```
fd1 = open("./test", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
fd2 = dup(fd1);
fd3 = open("./test", O_RDWR);
write(fd1, "++++", 4);
write(fd2, "----", 4);
lseek(fd1, 4, SEEK_SET);
write(fd3, "----", 4);
write(fd2, "++++", 4);
```

i) [0.75v] Que conteúdo guarda o ficheiro `./test` antes da execução da `lseek()`?

a. ----
b. ++++
c. ++++----
d. Nenhuma das anteriores

ii) [0.75v] Que conteúdo guarda o ficheiro `./test` depois da execução da última `write()`?

a. ++++----
b. ----++++
c. ++++
d. ----

Grupo 2 [Tarefas e processo, 1.2v]

1. [0.6v] Considere este programa:

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

int main(int argc, char **argv) {
    int pid, i;
    i=0;
    i++;

    pid=fork()

    if (pid==0) {
        i=i-1;
        printf("F:%d\n",i);
    }
    else if (pid>0) {
        i=i+1;
        printf("P:%d\n",i);
    }
}
```

Qual destes outputs pode ser produzido?

a. F:0
   P:1
b. F:0
   P:2
c. F:1
   P:0
d. Nenhum dos outputs acima é possível

2. [0.6v] Qual das seguintes afirmações é **falsa**?
   a. O código a seguir a chamada à `exec()` só é executado caso a chamada sistema falhar
   b. A tabela de ficheiros abertos do processo é limpa se a chamada à `exec()` tiver sucesso
   c. A área de dados e a pilha do programa atual são libertados caso a `exec()` tiver sucesso.

Indique d. caso todas as afirmações acima sejam verdadeiras.

Grupo 3 [Exclusão mútua, 3.5v]

1. [3.5v] Considere um programa de gestão de um cinema *single-threaded* em que o estado dos lugares do cinema é guardado no array (variável global):

```
int[MAX_LUG] lugares;
```

O valor 0/1 na entrada i deste array codifica respetivamente o estado livre/ocupado.
O programa disponibiliza a rotina:

```
int reserva(int N, int[] lug)
```

que tenta reservar um conjunto de *N* lugares, cujos índices são passados pelo array de inteiros *lug*. Esta rotina devolve 0 se conseguir adquirir todos os lugares, isto é se todos os lugares especificados em input são livres. Caso contrário, não adquire nenhum lugar e devolve -1.

O programa disponibiliza também a rotina:

```
void liberta(int N, int[] lug)
```

que marca como livres todos os lugares passados em input.

O código das duas rotinas é apresentado abaixo. Estenda o código destas duas rotinas para o tornar *thread-safe* utilizando apenas mutexes. A solução deve maximizar o paralelismo.
Ignore o tratamento dos erros e a inicialização dos mutexes.
Se for útil para simplificar a programação poderá utilizar a função `void int-sort(int size, int* array)` que permite ordenar um array de inteiros (de tamanho `size`) que lhe é passado por input

```
//Adicionar aqui a declaração de variáveis globais adicionais
int [MAX_LUG] lugares;
pthread_mutex_t locks[MAX_LUG];


void liberta(int N, int[] lug) {

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

            mutex_lock(&locks[lug[i]];

            lugares[lug[i]]=0;

            mutex_unlock(&locks[lug[i]];

      }

}


int reserva(int N, int[] lug) {
      int i;
      int j;

      int-sort(N, lug);
      for (i=0; i<N;i++) { //verifica primeiro se algum lugar estiver ocupado
            mutex_lock(&locks[lug[i]]);

            if ( lugares[lug[i]]==1 ) {

                  for (j=i; j=0; j--)
                    mutex_unlock(&locks[lug[j]])

                  return -1; //retornando se for este o caso
            }
      }


      // se não reserva os lugares
      for (i=0; i<N;i++) {

            lugares[lug[i]]=1;.
            mutex_unlock(&locks[lug[i]])

      }


return 0;
}
```

Grupo 4 [Semáforos, 2.4v]

1. [0.6v] Qual das seguintes observações é verdadeira?
   a) Para bloquear uma tarefa T1 até o acontecimento de um evento detetado pela tarefa T2, pode-se usar um semáforo inicializado com 0, onde T1 espera e T2 assinala.
   b) Para bloquear uma tarefa T1 até o acontecimento de um evento detetado pela tarefa T2, pode-se usar um semáforo inicializado com 1, onde T1 espera e T2 assinala.
   c) Para bloquear uma tarefa T1 até o acontecimento de um evento detetado pela tarefa T2, pode-se usar um semáforo inicializado com 0, onde T1 assinala e T2 espera.
   d) Nenhuma das anteriores

2. [1.8v] Considere a seguinte implementação do problema produtores-consumidores

```
/* criação dos objectos de sincronização */
pthread_mutex_t semExMut = PTHREAD_MUTEX_INITIALIZER; sem_init(&MsgparaLer,
0, 0);  sem_init(&bufferCheio, 0, N);

void EscreveMsg(t_msg Msg){
      sem_wait(&bufferCheio);
      pthread_mutex_lock(&semExMut);
      buffer[indiceEscrita] = Msg;
      indiceEscrita= (indiceEscrita+1)%N;
      pthread_mutex_unlock(&semExMut);
      sem_post(&MsgparaLer);
}

void LeMsg(t_msg *Msg) {
      sem_wait(&MsgparaLer);
       pthread_mutex_lock(&semExMut);
      *Msg = buffer[indiceLeitura];
      indiceLeitura=(indiceLeitura+1)%N;
      pthread_mutex_unlock(&semExMut);
      sem_post(&bufferCheio);
}
```

2.1 [0.6v] Considere trocar a ordem das linhas:

```
      sem_wait(&bufferCheio);
      pthread_mutex_lock(&semExMut);
```

na função `EscreveMsg()`. Qual das seguintes afirmações é verdadeira?

a) O programa é menos eficiente, mas funciona corretamente
b) O programa pode gerar interblocagem
c) Novas mensagens podem vir a ser escritas quanto não há espaço no buffer
d) O programa pode ser afetado pelo problema da míngua

2.2 [0.6v] Considere trocar a ordem das linhas:

```
pthread_mutex_unlock(&semExMut);
sem_post(&bufferCheio);
```

na função `LeMsg()`. Qual das seguintes afirmações é verdadeira?

a) O programa é menos eficiente, mas funciona corretamente
b) O programa pode gerar interblocagem
c) Novas mensagens podem vir a ser escritas quanto não há espaço no buffer
d) O programa pode ser afetado pelo problema da míngua

2.3 [0.6v] Qual das seguintes afirmações acerca do código acima é verdadeira:

a) Dado que leitores e escritores manipulam o mesmo buffer é necessário usar um trinco comum entre leitores e escritores
b) Seria possível aumentar o paralelismo usando um trinco para os leitores e outro para os escritores, dado que nunca podem aceder concorrentemente à mesma posição do buffer
c) Seria possível aumentar o paralelismo usando um trinco só na rotina `EscreveMsg`, pois a rotina `LeMsg` apenas lê do buffer.
d) Nenhuma das anteriores

Grupo 5 [Variáveis de condição, 1.5v]

1. [1.5v] Pretende-se implementar uma aplicação em que existem duas classes de tarefas: as *publishers* que publicam notícias (*posts*), e as *subscribers* que ficam a espera até surgir alguma noticia.
   As tarefas *publishers* podem usar a rotina `append_post(int index, char[] news)` para armazenar uma notícia no sistema e associá-la ao índice `índex`.
   As tarefas *subscribers* podem obter uma notícia usando a rotina `char[] obtain_post(int índex)` que devolve a noticia armazenada no sistema com índice `índex`.

   Preencha as "caixas" no código abaixo, com **zero**, **uma** ou **mais de que uma** instrução, entre as listadas na pergunta. A solução deverá usar variáveis de condição de forma a evitar espera ativa para as tarefas *subscribers*. Por simplicidade, supor que não existem limitações no sistema de armazenamento das notícias, o que garante que os *publishers* nunca precisam de se bloquear à espera que as notícias venham a ser "consumida" por um *subscriber*.

```
int posts = 0; pthread_mutex_t* m; pthread_cond_t* c;

void subscriber() {
     int read_posts=0;
     while (1) {
          [ f. ]
          [ a. ] (read_posts == posts) [ c. ]
          obtain_post(posts);
          read_posts++;
          [ g. ]
     }
}

void publisher(char[] message) {
     [ f. ]
     posts++;
     append_post(post, message);
     [ i. ou g. ]
     [ g. ou i. ]
}
```

a. `while`
b. `pthread_cond_wait(c)`
c. `pthread_cond_wait(c,m)`
d. `pthread_cond_wait(m)`
e. `if`
f. `pthread_mutex_lock(m);`
g. `pthread_mutex_unlock(m);`
h. `pthread_cond_signal(c);`
i. `pthread_cond_broadcast(c);`

Grupo 6 [Signals, 1.2v]

1) [0.6v] O mecanismo de *signals* é de grande importância para o tratamento de exceções desde o Unix inicial. Qual das seguintes afirmações é verdadeira?
   a) A rotina de tratamento do *signal* ou *sighandler é* uma rotina de interrupção instalada no núcleo
   b) O *signal* quando detetado coloca em execução uma *thread* no contexto do processo que irá executar o *sighandler*
   c) É uma rotina normal que fica referenciada na tabela de tratamento de *signals* do processo. Quando o processo vai passar de modo núcleo a modo utilizador, é testado se existe um *signal* pendente e por manipulação dos *stacks* núcleo e utilizador a rotina é posta a correr em modo utilizador
   d) O *signal* é tratado por uma rotina normal que fica referenciada na tabela de tratamento de *signals* do processo. Quando o processo passa de modo utilizador a modo núcleo, é testado se existe um *signal* pendente e por manipulação do *stack* a rotina é posta a correr em modo núcleo

2) [0.6v] Qual das seguintes afirmações é verdadeira?
   a) Um processo pode ignorar todos os signals
   b) Um processo pode definir uma rotina de tratamento para todos os signals
   c) Um processo pode ter associado um tratamento por omissão para todos os signals
   d) O signal SIGKILL apenas pode ser tratado em modo supertulizador

Grupo 7 [Pipes e comunicação, 1.2v]

1) [0.6v] Pretende-se criar um canal de comunicação entre um processo pai e um processo filho em Unix. Qual das seguintes afirmações é verdadeira?
   a) Para este canal de comunicação só se pode utilizar um pipe.
   b) Não é possível usar um pipe porque o processo filho não tem possibilidade de o abrir.
   c) Se o filho herda um pipe criado pelo pai apenas o pai pode enviar informação para o filho.
   d) As duas extremidades de um pipe criado pelo pai ficam automaticamente abertas no processo filho quando se efetuar o fork.

2) [0.6v] Qual das seguintes afirmações sobre Pipes e FIFOs (ou Pipes com nomes) é verdadeira?
   a) O pipe é unidireccional e o fifo é bidireccional.
   b) O pipe é bidireccional e o fifo é unidireccional
   c) São idênticos em termos de mecanismo de comunicação, mas diferem na atribuição de nome.
   d) A criação de um fifo é idêntica à de um ficheiro.

Grupo 8 [Despacho e escalonamento. 1.5v]

1) [1.5v] Simplificação do funcionamento do CFS. Considera tempo de execução como conceito semelhante ao *vruntime*. O sistema tem **preempção** e apenas um CPU core.

Processo A – bloqueado; tempo de execução 10
Processo B – Em execução; tempo de execução 20
Processo C – Executável; tempo de execução 22

Considere que o processo A é desbloqueado ao final de 5 unidades tempo de execução do processo B. Qual das seguintes afirmações é verdadeira?

a) O processo A vai para o estado executável e aguarda o final do *timeslice* do processo B
b) O processo A vai para o estado executável e retira o processo B de execução, voltando o processo B ao estado executável com tempo de execução 25
c) O processo A vai para o estado executável e retira o processo B de execução, voltando o processo B ao estado executável com o mesmo tempo de execução (isto é 20) uma vez que não concluiu o *timeslice*
d) O processo A vai para o estado executável e retira o processo B de execução. Ao fim de 2 unidades bloqueia-se novamente o processo B retoma a execução

Grupo 9 [Chamadas de sistemas Linux, 1.2v]

1) [0.6v] O processo executa o seguinte extrato de código

```
pid = fork ();
if (pid==0) {
    ….
    exit(0);
} else {
    ……
    pid = wait (&estado);
}
```

Qual das seguintes afirmações é verdadeira?

a) O processo pai bloqueia-se à espera que um filho termine. Só quando um terminar será desbloqueado. Portanto, é necessário ter cuidado na programação para não colocar o processo pai em espera se o filho já terminou
b) O processo pai bloqueia-se à espera que um filho termine. Se algum já está no estado Zombie/Defunct retorna imediatamente indicando o PID e o estado de terminação
c) O processo pai bloqueia-se à espera que um filho termine, mas se quando executa a função já terminaram todos os filhos. o `wait()` retorna com -1
d) O processo pai bloqueia-se à espera que um filho termine com o estado especificado no parâmetro. Só quando um processo filho nestas condições terminar, o pai será desbloqueado

2) [0.6v] Quando um processo chama `exit()`, qual das seguintes afirmações é verdadeira?
   a) A chamada sistema elimina as regiões de memória e a *task_struct* do processo.
   b) A chamada elimina as regiões de memória e a *task_struct* do processo. Fecha também os ficheiros abertos, podendo nessa situação detetar que um ficheiro já não tem nenhum hard link e consequentemente eliminando-o.
   c) A chamada elimina as regiões de memória e a *task_struct* do processo. Fecha também os ficheiros abertos, podendo nessa situação detetar que um ficheiro deveria ser apagado invocando *unlink* para o diretório que contém o ficheiro.
   d) A chamada elimina as regiões de memória do processo. Fecha também os ficheiros abertos e elimina-o caso detete que um ficheiro já não tem nenhum hard link nem outros processos ativos com o mesmo ficheiro aberto. Não elimina a *task_struct,* marcando o estado do processo como *defunct/zombie*

Grupo 10 [Gestão da Memória, 3.6v]

1) [0.6v] Num sistema com endereçamento virtual de 32 bits e 12 bits de deslocamento, o processo A gerou o endereço virtual 00030024 e o processo B no *time-slice* seguinte acedeu ao mesmo endereço. Qual das seguintes afirmações é verdadeira?
   a) É um erro dois processos terem endereços virtuais iguais provocando um conflito no acesso à memória
   b) É um erro os processos podem ter o mesmo endereço virtual, mas o TLB não ia distinguir que o processo foi comutado e provocaria um acesso errado
   c) Não é erro, os processos podem ter o mesmo endereço virtual. As bases das respetivas entradas na tabela de páginas somadas ao deslocamento é que definem o endereço real
   d) Não é erro, os processos podem ter o mesmo endereço virtual. Dado que a página foi usada pelo processo A é necessário substituí-la na memória física pela página do processo B, antes de este poder prosseguir

2) [1.5v] Considere um sistema de memória paginada, com um único nível de paginação, em que cada página tem 4K. Mais precisamente, o formato do endereço é:

Página (20 bits)	Deslocamento (12 bits)

Considere que um processo começa a sua execução e a TLB está limpa e que todas as páginas acedidas pelo processo se encontram carregadas em memória principal. Considere a seguinte sequência de acessos a endereços virtuais (todos os endereços estão em hexadecimal):

1. 0xA0001FB0
2. 0xA100F100
3. 0xA0001FB4
4. 0xA0001FB8
5. 0xA000D234
6. 0xA000D238
7. 0xA100F700
8. 0xA000D23A
9. 0xA100F800
10. 0xA0001FBA

2.a) [0,75v] Considere que a TLB está inicialmente limpo e possui espaço para traduzir 32 páginas e que além do TLB não existam outras caches no processador. Qual o *overhead* gerado pelo sistema de memória virtual na sequência de acessos descritos acima medido em número de acessos adicionais à memória física, em relação a um sistema usando endereçamento real.

   a) 10%
   b) 20%
   c) 40%
   d) 30%

Notas sobre a resolução

2a) A informação relevante para o exercício é as páginas acedidas. O formato do endereço virtual indica que os 20 bits mais significativos são o número da página (5 algarismos hexadecimais) e os 12 bits menos significativos o deslocamento (3 algarismos hexadecimais)

Com base nesta divisão poderíamos contruir a seguinte tabela

Endereço virtual	Numero de pagina	Acesso a memória	Presente no TLB	Número total de acessos a memória
1 0xA0001FB0	A0001	2 acessos – tabela de páginas e página	não	2
2. 0xA100F100	A100F	2 acessos – tabela de páginas e página	não	4
3. 0xA0001FB4	A0001	1 acesso á página	Sim	5
4. 0xA0001FB8	A0001	1 acesso á página	Sim	6
5. 0xA000D234	A000D	2 acessos – tabela de páginas e página	não	8
6. 0xA000D238	A000D	1 acesso á página	Sim	9
7. 0xA100F700	A100F	1 acesso á página	Sim	10
8. 0xA000D23A	A000D	1 acesso á página	Sim	11
9. 0xA100F800	A100F	1 acesso á página	Sim	12
10. 0xA0001FBA	A0001	1 acesso á página	Sim	13

Portanto 13 acessos à memória para resolver 10 endereços virtuais:
13-10/10= 30%

2.b) [0.75v] Responda à mesma pergunta num sistema com dois níveis de paginação, com a seguinte estrutura de endereços:

Página (10 bits)	Página (10 bits)	Deslocamento (12 bits)

a) 40%
b) 50%
c) 60%
d) 70%

Notas sobre a resolução

A diferença principal é que existindo dois níveis de páginas haverá 3 acessos à memória em caso de TLB miss:

1. Acesso à TP de nível 1 para determinar o endereço da tabela de página de nível 2,
2. Acesso à TP de nível 2 para determinar o endereço real da página
3. Acesso efetivo à página

Nota para representar na tabela os endereços coloca-se o problema de representar os 10 bits que seriam um algarismo hexadecimal ente 0-3 seguido de dois números hexadecimais como existem poucas páginas a transformação de endereços era relativamente fácil – A00001 – 280-001; A1000f – 284-00F; A0000D – 280 00D

Endereço virtual	Numero de pagina nível 1 – 10 bits	Número de página nível 2- 10 bits	Acesso a memória	Presente no TLB	Número total de acessos a memória
1 0xA0001FB0	280 (A00)	001	3 acessos – TP de nível 1 e TP de nível 2 e página	não	3
2. 0xA100F100	284 (A10)	00F	3 acessos – TP de nível 1 e TP de nível 2 e página	não	6
3. 0xA0001FB4	280 (A00)	001	1 acesso á página	Sim	7
4. 0xA0001FB8	280 (A00)	001	1 acesso á página	Sim	8
5. 0xA000D234	280 (A00)	00D	3 acessos – TP de nível 1 e TP de nível 2 e página	Não	11
6. 0xA000D238	280 (A00)	00D	1 acesso á página	Sim	12
7. 0xA100F700	284 (A10)	00F	1 acesso á página	Sim	13
8. 0xA000D23A	280 (A00)	00D	1 acesso á página	Sim	14
9. 0xA100F800	284 (A10)	00F	1 acesso á página	Sim	15
10. 0xA0001FBA	280 (A00)	001	1 acesso á página	Sim	16

Portanto 16 acessos à memória para resolver 10 endereços virtuais:

16-10/10= 60%

3) [1.5v] Considere que na execução do Processo a sua tabela de páginas (simplificada) era a descrita em seguida e que o TLB tinha as seguintes entradas. O sistema tem um endereçamento virtual com 32 bits e páginas de tamanho de 4 kBytes

Tabela de Páginas

Presente	Page number	Base – frame number	Protecção
S	00000	20000	r-x
S	00001	01C00	r-x
N	00002	-	rw-
S	00003	00100	rw-

TLB

Validade	Page number	Base – frame number	Protecção
S	00001	01C00	r-x
N	00120	20000	r-x
S	00000	20000	r-x
S	00003	00100	rw-

Considere que o programa gera os seguintes acessos (na ordem especificada pela tabela). Preencha todos os campos relevantes da tabela da consequência da tradução do endereço. Para resolver o problema poderá eventualmente necessitar de novas page frames, poderá arbitrar um valor ou por exemplo usar 0x00300 ou 0x30000.

Endereço	Tipo de acesso	Endereço físico	Resolvido no TLB	Resolvido na tabela de páginas	Page Fault ou Acesso invalido (especificar)
00000100	execução	20000100	Sim	-	Não
00002150	escrita	00300150	Não	Sim	Page Fault
00002120	escrita	00300120	Sim	-	Não
00004150	execução	-	-	-	Acesso invalido (página não existente)
00002010	leitura	00300010	Sim	-	Não
00001000	Escrita	O1C00000	Sim	-	Acesso inválido (proteção)