LEIC/LETI – 2021/22 – 2º Exame de Sistemas Operativos

26 de fevereiro de 2022, 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 mais 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 I [Programação com processos, 2v]

Considere o seguinte programa:

```
int main() {
  pid_t pid_fork = fork ();
  printf("pid_fork=%d, my_pid=%d\n", pid_fork, getpid());

  sleep(100);

  int w = wait(NULL);
  printf("wait devolveu %d\n", w);

  exit(0);
}
```

O programa acima foi executado por um processo com PID=1015. Durante essa execução, foi criado um processo filho com PID=1016.

1. [1v] Nos primeiros instantes de execução, qual dos seguintes *outputs* foi apresentado?

   a. pid_fork=1015, my_pid=1015
      pid_fork=1016, my_pid=1016
      (ou ambas as linhas na ordem inversa)
   b. pid_fork=1016, my_pid=1015
      pid_fork=0, my_pid=1016
      (ou ambas as linhas na ordem inversa)
   c. pid_fork=1016, my_pid=1016
      pid_fork=0, my_pid=0
      (ou ambas as linhas na ordem inversa)
   d. Nenhuma das anteriores

2. [1v] Nos últimos instantes de execução, qual dos seguintes *outputs* foi apresentado?

   a. wait devolveu 1016
      wait devolveu -1
      (ou ambas as linhas na ordem inversa)

b. wait devolveu 1015
wait devolveu 1016
(ou ambas as linhas na ordem inversa)
c. wait devolveu 0 (em duplicado)
d. Nenhum *output* é apresentado.

Grupo II [Sistema de ficheiros, 3v]

1. [1v] Qual das seguintes afirmações é **falsa**:
   1. No sistema de ficheiro CP/M o tamanho máximo dos ficheiros é limitado pelo tamanho das entradas dos diretórios
   2. A utilização de blocos de tamanho fixo em sistemas de ficheiros permite evitar fragmentação externa
   3. A utilização de blocos de tamanho fixo em sistemas de ficheiros permite evitar fragmentação interna
   4. No FAT-16 o tamanho máximo de um ficheiro é de 2^16 - 1 blocos

2. [1v] Qual das seguintes afirmações é **falsa**:
   1. Ao apagarmos um hard-link o ficheiro correspondente é sempre apagado
   2. No Ext3 os diretórios guardam a associação entre os nomes do ficheiros e os relativos i-nodes (i-number)
   3. O número de ficheiros guardados num sistema de ficheiros ext3 não pode ultrapassar o número de entradas na tabela de i-nodes
   4. No Ext3 quer os ficheiros quer os diretórios têm um i-node associado que descreve os respeitivos metadatos.

3. [1v] Suponha que dois processos, P1 e P2, inicialmente sem nenhum ficheiro aberto (além de stdin, stdout, stderr), executam as seguintes chamadas:

   P1:
   fd1 = open (“/tmp/fileA”, O_WRONLY);

   P2:
   fd2 = open (“/tmp/fileA”, O_RDONLY);

   O ficheiro "/tmp/fileA" é um ficheiro normal, armazenado em disco por um sistema de ficheiros.

   Qual das seguinte afirmações é verdadeira?

   a. Se um dos processos tiver sucesso na chamada open, ou outro garantidamente não tem sucesso na sua chamada.

   b. O valor de fd1 e fd2, retornado por cada chamada, é garantidamente diferente, pois apontam para entradas diferentes nas tabelas de ficheiros abertos.

   c. Ambas as chamadas, caso tenham sucesso, criam entradas distintas na tabela de ficheiros abertos global do sistema.

   d. Quando P1 chama open para escrever, a função bloqueia até P2 chamar open para ler; e vice-versa.

Grupo III [Tarefas e sincronização em memória partilhada, 6,5v]

Considere a seguinte implementação de operações sobre contas bancárias mantidas em memória partilhada.

```
typedef struct {
  int saldo;
  int numMovimentos;
/* etc. */
} conta_t;


int depositar_dinheiro(conta_t* conta, int
valor) {
  if (valor < 0)
    return -1;
  conta->saldo += valor;
  conta->numMovimentos ++;
  return valor;
}
```

```
int levantar_dinheiro(conta_t* conta, int valor) {
  if (valor < 0)
    return -1;
  if (conta->saldo >= valor) {
    conta->saldo -= valor;
    conta->numMovimentos ++;
  }
  else
    valor = -1;
  return valor;
}

void consultar_conta(conta_t* conta) {
  int s, n;
  s = conta->saldo;
  n = conta->numMovimentos;
  printf("saldo=%d, #movimentos=%d \n", s, n);
}
```

1. [2v] Assuma duas tarefas, A e B, criadas nesta ordem e que se executam concorrentemente. Ambas recebem como argumento um ponteiro para uma dada conta, que inicialmente tem *saldo* e *numMovimentos* nulos.
   A tarefa A executa a função fnA (deposita 10, deposita 20, deposita 30, termina), enquanto que a tarefa B executa a função fnB (tenta repetidamente levantar 5, terminando assim que tiver sucesso).
   Assim que ambas as tarefas terminem, a tarefa *main* imprime o estado da conta invocando a função consultar_conta() e termina.
   Complete o código seguinte para implementar este programa concorrente. Nota que poderá ser necessário adicionar instruções e não apenas completar as instruções parcialmente especificadas.

```
void *fnA (void *arg) {
  conta_t* conta = __(conta_t*) arg___ ;
  depositar_dinheiro (conta, 10);
  depositar_dinheiro (conta, 20);
  depositar_dinheiro (conta, 30);
  return NULL;
}

void *fnB (void *arg) {
  conta_t* conta = ___ (conta_t*) arg ___ ;
  while (levantar_dinheiro (conta, 5) == -1) {}
  return NULL;
}

int main() {
  pthread_t tidA, tidB;
  conta_t *c = obterConta(ID_CONTA_EXEMPLO); //Função auxiliar que obtém a conta


  pthread_create (&tidA, 0, _fnA _, __c_);



  pthread_create (&tidB, 0,_fnB__, __c_);


  pthread_join(tidA,NULL);
  pthread_join(tidB,NULL);

  consultar_conta(c);



  exit(EXIT_SUCCESS);
}
```

2. [1v] Quando executado numa máquina *single-core*, o programa paralelo acima demora muito mais tempo a concluir do que uma variante sequencial que simplesmente executa fnA() e fnB() nesta ordem. Qual a razão?

   a. Como as secções críticas não estão sincronizadas com trincos, o programa é mais lento.
   b. A execução paralela causa mais faltas de página.
   c. Uma parte substancial é gasta em espera ativa.
   d. Nenhuma razão. Aliás, o programa paralelo é mais rápido que a variante sequencial.

3. Executou-se o programa descrito na alínea anterior várias vezes e observou-se a linha impressa no ecrã no final. Para cada *output* indicado abaixo, qual ou quais são possíveis (mesmo que não aconteçam sempre)..
   Em caso de resposta errada, desconta 100% da cotação da alínea.

   a. [0,5v] saldo=55, #movimentos=4 Possível [X] Impossível [ ]
   b. [0,5v] saldo=55, #movimentos=3 Possível [X] Impossível [ ]
   c. [0,5v] saldo=25, #movimentos=4 Possível [X] Impossível [ ]

4. [2v] Pretende-se a sincronização necessária ao código das operações *levantar_dinheiro*, e *consultar_conta* (a função depositar_dinheiro fica excluída deste exercício). Adicione as primitivas de sincronização necessárias no código disponibilizado na próxima folha.
   - Pode recorrer a trincos lógicos (*pthread_mutex*) e/ou a trincos de leitura-escrita (*pthread_rwlock*). Não se esqueça de o(s) declarar; pode omitir a sua inicialização.
   - Deve maximizar o paralelismo, tendo em conta que (i) pode haver um evelado número de tarefas a chamar concorrentemente ambas as operações; (ii) entre ambas as operações, a mais frequente é a *consultar_conta.*

```
typedef struct {
  int saldo;
  int numMovimentos;
  pthread_rwlock_t lock;

} conta_t;



int levantar_dinheiro(conta_t* conta, int valor) {


  if (valor < 0)


    return -1;


  pthread_rwlock_wrlock(&conta->lock);

  if (conta->saldo >= valor) {


    conta->saldo -= valor;


    conta->numMovimentos ++;


  }
  else


    valor = -1;

  pthread_rwlock_unlock(&conta->lock);
  return valor;
}

void consultar_conta(conta_t* conta) {
  int s, n;

  pthread_rwlock_rdlock(&conta->lock);

  s = conta->saldo;


  n = conta->numMovimentos;

  pthread_rwlock_unlock(&conta->lock);


  printf("saldo=%d, #movimentos=%d \n", s, n);


}
```

Grupo IV [Comunicação entre processos, 2v]

1. [2v] Pretende-se implementar um programa que execute "/bin/ls -ltr | grep SO", usando o processo filho para correr o primeiro executável (ls) e o pai para o segundo executável (grep).

```
int main() {
        int fds[2]; int p;
        1.h,g
        if (p > 0) {
            2. d,f, i
        }
        else if (p == 0) {
            3. b,e, c
        }
        4.a
}
```

Preencha as "caixas" no código de cima, com **zero**, **uma** ou **mais de que uma** instrução, entre as listadas em baixo.

Notas:

- Caso indique mais do que uma instrução para uma dada caixa, deverá indicá-las na ordem em que se pretendem sejam executadas, separadas por virgulas (p.e., indicar "c,d" se se pretende executar primeiro *exec* para ls e a seguir o close(0)).
- Não precisa usar todas as instruções listadas abaixo.

a. return EXIT_FAILURE;
b. close(1);
c. execl("/bin/ls", "ls", "-ltr", 0);
d. close(0);
e. dup(fds[1]);
f. dup(fds[0]);
g. p=fork();
h. pipe(fds)
i. execl("/usr/bin/grep", "grep", "SO", 0);
j. wait(NULL);

Grupo V [Memória Virtual, 3,5v]

1. [1v] Qual das seguintes afirmações é **verdadeira**:
   a. Usar um quantum menor implica "resets" mais frequentes do TLB para processos "CPU-bound"
   b. Usar um quantum maior implica "resets" mais frequentes do TLB para processos "CPU-bound"
   c. TLBs de tamanho maior devem ser "reset" mais frequentemente
   d. Páginas de tamanho maior permitem reduzir a frequência de reset do TLB

2. [1v] Qual das seguintes afirmações é **falsa**:
   a. O mecanismo de Copy-On-Write permite acelerar a execução da chamada de sistema fork()
   b. O bit Copy-On-Write é escrito pelo hardware quando é detetada uma exceção de violação de acesso (falta de permissões de escrita) numa página.
   c. O bit Copy-On-Write é lido pelo SO quando o hardware levanta uma exceção de violação de acesso (falta de permissões de escrita) numa página.
   d. Se houver uma exceção de violação de permissões de acesso à pagina e o bit Copy-On-Write tem valor 1, o OS copia a página.

3. [1,5 val] Considere um sistema de gestão de memória com 16 bits de espaço de endereçamento virtual, com páginas de 256B e onde existem dois processos, P1 e P2.
   Suponha que os processos P1 e P2 tenham as seguintes tabelas de páginas:

   P1:

Página	Presente	Protecção	Base
0	0	RW	
1	1	R	0x02
2	1	R	0x04

   P2 :

Página	Presente	Protecção	Base
0	1	RW	0x02
1	0	R	
2	1	R	0x01

   Na tabela abaixo, indique qual é o endereço físico correspondente aos seguintes endereços virtuais, caso possível. Caso o acesso der origem a alguma exceção, indique também o tipo da exceção.
   Assuma que, em caso de falta de páginas, o SO aloca tramas (*page frames*) livres a partir do endereço 0x09 00.

Processo	Acesso	End. Virtual	End. Físico	Eventuais exceções
P1	Leitura	0x01 22	0x0222	
P2	Escrita	0x00 22	0x0222	
P1	Leitura	0x03 01	-	Endereço inválido
P1	Leitura	0x00 FF	0x09ff	Page fault

Grupo VI (Chamadas de Sistema e Escalonamento, 3v)

1. [1v] Qual das seguintes afirmações é **verdadeira**:
   a. As chamadas de sistema são despoletadas por interrupções hardware
   b. Antes de ativar a rotina de tratamento de um *signal*, o núcleo guarda na pilha núcleo o endereço da próxima instrução a ser executada pelo programa que recebe o signal.
   c. Durante a execução da chamada de sistema *execl* o núcleo copia os argumentos de input da *execl* da pilha utilizador para a pilha núcleo
   d. Um processo permanece no estado zombie indefinidamente se o processo pai termina sem invocar wait()

2. [1v] Qual das seguintes afirmações é **falsa**:
   a. Um processo no estado executável executa sempre em modo núcleo antes de executar em modo utilizador.
   b. A chamada de sistema *nice* permite aumentar a prioridade apenas aos utilizadores privilegiados (superuser)
   c. No algoritmo de escalonamento de Unix (apresentado nas teóricas) valores negativos da prioridade correspondem a valores mais elevados de prioridade efetiva.

   Caso nenhuma das afirmações anteriores seja falsa, indicar d.

d

3. [1v] No algoritmo de escalonamento de Linux (apresentado nas teóricas) que usa quantum variáveis e épocas, o quantum máximo atribuído a um processo que está bloqueado indefinidamente tende a ser:
   a. 2 vezes o quantum mínimo
   b. 3 vezes o quantum mínimo
   c. 4 vezes o quantum mínimo
   d. Nenhuma das anteriores