LEIC/LETI – 2021/22 – 1º Exame de Sistemas Operativos

15 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, 3v]

Considere o seguinte programa, que chama iterativamente uma função (f) com K diferentes argumentos e, no final, indica quantas dessas chamadas tiveram sucesso.

```
int main () {
  int success = 0;
  for (int i = 0; i<K; i++) {
    if (f(i) == 0)
      //This was a successful call
      success ++;
  }
  printf("successful calls: %d\n", success);
}
```

Devido a algum erro no código da função f, quando esta é chamada com alguns valores específicos como argumento, o código da função desencadeia alguma instrução ilegal (e.g., divisão por zero ou acesso a endereço inválido) que leva o processo a ser terminado abruptamente.

1. [2v] Pretende-se que, mesmo que a situação mencionada acima ocorra para algumas chamadas de f, o ciclo principal (apresentado acima) não termine e continue para a próxima iteração. Dessa forma, o programa iria continuar a contar as chamadas a f que completaram com sucesso.
   Estenda o programa acima de forma a cumprir esse requisito, tendo em conta estes requisitos:
   - Execute cada chamada a f dentro de um processo filho criado usando a função *fork*. No entanto, note que, tal como no programa original, cada iteração do ciclo só deve ser executada depois da iteração anterior ter sido executada (com ou sem sucesso).
   - Depois de chamar a função *wait*, pode assumir que a macro *EXITED_ZERO(status)* devolve verdadeiro caso o filho (cujo *exit status* é passado no argumento da macro) tenha terminado chamando *exit(0)*.

```
int main () {
  int success = 0;
  for (int i = 0; i<K; i++) {

    if (fork() == 0) {
      exit(f(i));
    }
    else {
      wait(&status);
      if (EXITED_ZERO(status))
        success++;
    }

  }
  printf("successful calls: %d\n", success);
}
```

2. [1v] Comparando o programa original com a variante pedida na alínea anterior, que **desvantagem** encontra na última?
   a. Usando processos filho, a nova variante só funciona em máquinas multi-core ou multiprocessador.
   b. A nova variante é menos segura pois cada processo filho pode correr com um UID diferente do processo inicial.
   c. Caso todas as K chamadas a f completem com sucesso, a nova variante é mais demorada que o programa original.
   d. A nova variante não permite que a função f aceda a ficheiros.

Grupo II [Sistema de ficheiros, 3v]

1. [2v] Considere um disco com uma capacidade total de 1GB formatado com um sistema de ficheiros FAT, onde o tamanho de cada bloco de dados é de 64KB, as referências para blocos de dados ocupam 32 bits e o espaço reservado para a *File Allocation Table* é de 128MB. Qual é a capacidade máxima efetivamente disponível para os utilizadores do sistema de ficheiros?
   a. 896MB
   b. 512MB
   c. 256MB
   d. Nenhuma das anteriores

0. [1v] Considere o excerto do seguinte programa:

```
int main() {
      int f1 = open("/home/alice/so/f.txt", O_RDONLY);
      //Restante programa omitido
}
```

Quantos i-nodes são acedidos no ext3 durante a execução deste excerto, supondo que a pasta /home/alice existe e que a pasta /home/alice/so não existe?
   a. 3
   b. 4
   c. 5
   d. Nenhuma das anteriores

Grupo III [Tarefas e sincronização em memória partilhada, 4v]

[4v] Considere um programa multi-tarefa composto pela tarefa principal (que executa a função fnMain) e, adicionalmente, N tarefas trabalhadoras (que executam a função fnWorker).
Uma versão simplificada destas funções está apresentada abaixo.
Resumidamente:

- A tarefa principal recebe pedidos de clientes (através de algum canal de comunicação).
- Cada pedido é um par *{int w, int v}*, em que *w* é um número que identifica a tarefa trabalhadora (entre 0 e N-1) responsável por processar o valor contido no pedido, *v*. O valor contido num pedido é sempre diferente de zero.
- Existe um vetor partilhado, value[N], cujas entradas têm o valor zero para indicar que tarefas estão vazias.
- Após receber um pedido {w,v}, a tarefa principal verifica se a entrada associada a *w* no vetor value está vazia. Se sim, a tarefa principal preenche essa entrada com o valor recebido; se não, descarta este pedido.
- Cada tarefa trabalhadora espera até que haja algum valor colocado na entrada respetiva no vetor. Assim que tal aconteça, a tarefa lê o valor, processa-o (uma operação que pode ser muito demorada) e, finalmente, liberta a sua entrada no vetor para permitir que a tarefa principal lhe passe pedidos posteriores.

Estenda o código abaixo com a sincronização das secções críticas e coordenação entre tarefas, que estão em falta. Idealmente, **maximize o paralelismo** e evite recorrer a espera ativa.

Na sua solução, pode recorrer a *mutexes (pthread_mutex)* e/ou variáveis de condição (*pthread_cond*). Não se esqueça de o(s) declarar (não precisa o(s) inicializar).

Nota: diferentes soluções são possíveis. De seguida apresentamos apenas uma delas.

```
/* Variáveis globais */
int value[N] = {0,...,0};

pthread_mutex_t m[N];
pthread_cond_t c[N];

fnMain() {

  int w, v;

  while (1) {

    /* receiveRequest coloca em w o id da
       tarefa a ativar e retorna o valor
       para processar */
    v = receiveRequest (&w);


    if (pthread_mutex_trylock(&m[w]) != 0)
      //A tarefa w ainda não libertou o valor
      //anterior, logo descartamos este novo
      //valor
      continue;

    if (value[w] == 0) {

      value[w] = v;

      pthread_cond_signal(&c[w]);

    }

    pthread_mutex_unlock(&m[w]);

  }

}
```

```
fnWorker(int wid) {


  while (1) {

    pthread_mutex_lock(&m[wid], &c[wid]);

    while(value[wid]==0)
      pthread_cond_wait(&c[wid], &m[wid]);


    /* Processa o valor fornecido. Muito demorada! */
    process (value[wid]);

    value[wid] = 0;

    pthread_mutex_unlock(&m[wid]);

  }


}
```

Grupo IV [Comunicação entre processos, 2v]

```
void apanhaCTRLC (int s) {
  char ch;
  printf("Quer de facto terminar a execucao?\n"); ch = getchar();
  if (ch == 's') exit(0);
  else {
    printf ("Entao vamos continuar\n");
    signal (SIGINT, apanhaCTRLC);
  }
}

int main () {
  signal (SIGINT, apanhaCTRLC);
  printf("Associou uma rotina ao signal SIGINT\n");
  for (;;)
    sleep (10);
}
```

[1v] Analise o programa acima apresentado. Pode-se afirmar que este programa foi desenvolvido para sistemas operativos cuja implementação dos *signals* oferece uma semântica especifica? Se sim, indicar qual.

System V

[1v] Supondo que o programa seja colocado em execução num sistema operativo com semântica System V, o que acontece se o utilizador carregar duas vezes de seguida na tecla CTRL-C enquanto o programa executa?
Suponha também que o processo corre em "foreground" na *shell* (de forma que sejam gerados dois SIGINT para esse processo).

É impresso "Quer de facto terminar a execucao?" na consola e de seguida o processo termina.

Grupo V [Memória Virtual, 4v]

1. Considere um sistema de gestão de memória baseado em segmentos, com 32 bits de espaço de endereçamento virtual, com 3 bits (os mais significativos) para a identificação do segmento.

   a. [1 Val] É possível que o número *total* de segmentos presentes na memória principal, numa dada altura, da maquina possa ser superior a 8?

   ☒ Sim ☐ Não

   O desconto em caso de resposta errada é -1v (100% da cotação prevista).

   b. [2 val] Suponha que existem 3 segmentos, S_a, S_b e S_c (a restante memória primária encontra-se livre):

- S_a: dimensão: 16kB alocado em memória principal no endereço base 0x0000 0000
- S_b: dimensão: 1MB, que se encontra carregado em memória principal no endereço base 0x0010 0000 (isto é 2^20 em binário)
- S_c: dimensão: 32kB, que não se encontra em memória principal

Preencha a tabela de segmentos de um processo, P1, cujo espaço de endereçamento contém os três segmentos indicados acima, onde:

- S_a é mapeado como o segmento 1 do processo P1 com permissões de leitura e escrita.
- S_b é mapeado como o segmento 2 do processo P1 com permissão de leitura.
- S_c é mapeado como o segmento 0 do processo P1 com permissão de leitura.

Segment ID	P	Prot	Limite	Base
0	**0**	**R**	**32 kB**	**-**
1	**1**	**RW**	**16 kB**	**0x0000 0000**
2	**1**	**R**	**1 MB**	**0x0010 0000**

2. [1 val] Qual das seguintes afirmações sobre paginação é **falsa**:
   a. Usar páginas de tamanho maior permite reduzir o tamanho das tabelas de páginas.
   b. As tabelas de páginas multinível permitem reduzir o tempo necessário para a tradução de endereços virtuais, quando comparadas com tabelas de páginas com apenas um nível.
   c. O TLB deve ser limpo pelo SO quando é efetuada a comutação entre processos diferentes.
   d. Os sistemas baseados em paginação não sofrem de fragmentação externa.

   ☐

Grupo VI (Chamadas de Sistema e Escalonamento, 4v)

1. [1v] Qual das seguintes afirmações é **falsa**:
   a. O *núcleo* de um sistema operativo é ativado apenas caso ocorra uma interrupção causada pelo *hardware*.
   b. Se o núcleo não utilizasse uma pilha (*stack*) diferente da usada pelas aplicações, aplicações maliciosas poderiam manipular o estado interno do *kernel.*
   c. A última instrução executada pelo despacho é *Return from Interrupt (RTI).*
   d. A comutação entre processos implica custos maiores do que a comutação entre tarefas do mesmo processo.

2. [1v] Qual das seguintes afirmações é **falsa**:
   a. Os processos "I/O bound" costumam não utilizar todo o *quantum* que lhe é disponibilizado.
   b. Os processos "CPU bound" são normalmente penalizados pelos algoritmos de escalonamento que utilizam prioridades dinâmicas.
   c. Ao aumentar do número de processos, a percentagem de tempo de CPU utilizada pelo algoritmo de escalonamento Unix (apresentado nas teóricas) tende a subir.
   d. Ao executar a chamada sistema *exit(...),* o SO liberta toda a memória associada ao contexto do processo (que invoca exit).

1. [2v] Suponha que o algoritmo baseado em épocas do escalonador Linux foi configurado para usar *quantum_base=10ms* e que existem apenas dois processos, P1 e P2, que são lançados simultaneamente com a mesma *prio_base* e *nice=0*, numa máquina equipada com apenas um CPU-core. Suponha também que, quando um processo se inicia, *quantum_por_utilizar_epoca_anterior=0*, e que:
   - o P1 é um processo *CPU-intensive* que i) gasta sempre todo o seu quantum, ii) é o primeiro a receber execução, e iii) nunca termina.
   - o P2 bloqueia-se depois de executar durante 6 ms; 8ms depois volta a desbloquar-se (isto é, fica executável); executa durante 11ms e termina.

   Quanto demora em tempo real a execução do P2?

   a. 34 ms
   b. 35 ms
   c. 36 ms
   d. Nenhuma das anteriores