Versão: v0 (deve corresponder ao algarismo menos significativo do número de aluno)

LEIC/LETI – 2020/21 – Repescagem do 1º Teste de Sistemas Operativos

8 de fevereiro de 2021, Duração: 1h00m

- Confira as instruções fornecidas no site da disciplina sobre a realização online desta prova.
- 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, 7v]

Pretende-se desenvolver um programa que recebe os seguintes argumentos (de linha de comando):

```
executável arg1 arg2 ... argn
```

Ao receber os argumentos acima, o ficheiro executável indicado no 1º argumento deve ser executado com cada um dos argumentos seguintes, de forma **sequencial**. Ou seja:

```
executável arg1
executável arg2
...
executável argn
```

Considere a seguinte implementação deste programa (que tanto pode ser correta como errada).

```
int main(int argc, char **argv) {
  int i;

  //Executa executável (argv[1]) com cada arg. (argv[2], argv[3], argv[4], etc.)
  for (i=2; i<argc; i++)
    execl(argv[1], argv[1], argv[i], NULL);

  exit(EXIT_SUCCESS);
}
```

1. [1,5v] Assuma que o programa é chamado com os argumentos “xpto a b c” e que todas as chamadas a execl são bem sucedidas.
   O comportamento observado será o que está especificado acima?
   A. Sim.
   B. Não, pois “xpto a”, “xpto b” e “xpto c” irão correr em paralelo.
   C. Não, pois só executará o primeiro passo (“xpto a”) e não executará os restantes.
   D. Não, pois a função execl só pode ser chamada por um processo filho.

2. [2,5v] Considere agora que se pretendia implementar uma variante do programa especificado no início do enunciado, mas com uma diferença: os comandos devem executar **em paralelo**, cada um num processo filho.
   Apresente uma solução (i.e., apresente a função main que implementa esta variante).

```
int main(int argc, char **argv) {
  int i;

  for (i=2; i<argc; i++) {
    if (fork() == 0)
      execl(argv[1], argv[1], argv[i], NULL);
  }

  for (i=2; i<argc; i++)
    wait(NULL);

  exit(EXIT_SUCCESS);
}

//Nota: como indicado no enunciado, na solução acima omitiu-se o
//tratamento dos retornos de erro das funções fork, execl, wait.
```

3. [1,5v] O output que os *n* processos filho criados pelo programa especificado na questão anterior enviam para o seu *stdout*:
   A. É descartado pelo sistema operativo, pois são processos filho.
   B. É apresentado na mesma consola do processo pai, pois cada processo herda o mesmo ficheiro *stdout* que o processo pai tinha.
   C. É passado ao processo pai pelo seu *stdin* através de um pipe.
   D. É retornado ao processo pai quando o filho chama exit.

4. [1,5v] Considere o seguinte programa *count*:

```
int contador_global = 0;
int main(int argc, char **argv) {
  int x = 0;
  if (argc==2)
    x = atoi(argv[1]);

  while (x>0) {
    contador_global ++;
    x --;
    sleep(1);
  }
  printf("%d.", contador_global);
}
```

   Assuma que o programa especificado na questão 2 era executado com os argumentos *count 100 200 300.* Assuma também que todas as chamadas sistema (fork, execl, etc.) têm sucesso. Qual o *output* esperado?
   A. É sempre este: 100.200.300.
   B. Tanto pode ser 100.200.300. como uma qualquer permutação destes valores (por exemplo, 100.300.200. ou 200.100.300. ou etc.).
   C. São 3 inteiros, cada um seguido por um ponto, mas como há acessos concorrentes ao contador global é incerto qual o valor que será impresso por cada processo.
   D. É sempre 300.300.300.

Versão: v0 (deve corresponder ao algarismo menos significativo do número de aluno)

Grupo II [Tarefas e sincronização em memória partilhada, 13v]

Considere o seguinte excerto de uma implementação concorrente de uma tabela de dispersão (*hash table*) que guarda elementos do tipo *element_t*.
Cada elemento tem uma chave (que o identifica) e um valor.

```
list_t hashtable[N];
pthread_mutex_t mutex[N];

void ht_init() {
  int i;
  for (i=0; i<N; i++) {
    hashtable[i] = new_empty_list();
    pthread_mutex_init(&mutex[i], 0);
  }
}

void ht_insert(element_t *e) {
  int h = hash(e->key);
  pthread_mutex_lock(&mutex[h]);
  list_insert(hashtable[h], e);
  pthread_mutex_unlock(&mutex[h]);
}

void ht_delete(key_t k) {
  int h = hash(k);
  pthread_mutex_lock(&mutex[h]);
  list_delete(hashtable[h], k);
  pthread_mutex_unlock(&mutex[h]);
}

element_t *ht_lookup(key_t k) {
  element_t *e;
  int h = hash(k);
  pthread_mutex_lock(&mutex[k]);
  e = list_lookup(hashtable[h], k);
  pthread_mutex_unlock(&mutex[h]);
  return e;
}

//Nota: definição dos tipos de dados (list_t, element_t, key_t) e funções
auxiliaries (hash, list_*, etc.) omitidos por simplicidade
```

1. [3,3v] Pretende-se construir um programa que:
   - Lança 4 tarefas (*pthreads*) escravas, cada uma identificada por um identificador 0, 1, 2 e 3.
   - Assim que todas as tarefas terminem o seu trabalho, o programa termina.
   - Cada tarefa escrava executa a função *void *work(int *)*, que recebe como argumento um ponteiro para o identificador inteiro da tarefa. Assuma que a função *work* já está implementada.

   Programe a função main deste programa. Não se esqueça de inicializar a tabela de dispersão (chamando a função *ht_init()* implementada no código acima).

```
#define NTHREADS 4

int main(int argc, char **argv) {
  int i;
  pthread_t threads[NTHREADS];
  int id[NTHREADS];

  ht_init();

  for (i=0; i<NTHREADS; i++) {
    id[i] = i;
    pthread_create(&threads[i], 0, work, &id[i]);
  }

  for (i=0; i<NTHREADS; i++)
    pthread_join(threads[i], NULL);

  exit(EXIT_SUCCESS);
}
//Tal como indicado no enunciado, omite-se o tratamento dos retornos
//de erro das funções pthread_create e pthread_join
```

2. [1,5v] Considere uma variante da implementação da tabela de dispersão em que só se usava **um único mutex** para sincronizar todos os acessos. Esta mudança pode piorar o desempenho.

   Considere os seguintes cenários:
   - Cenário A: 4 tarefas escravas
   - Cenário B: 24 tarefas (em máquina com 24 cores disponíveis)

   Em qual cenário espera que se observe maior degradação de desempenho?
   A. Maior degradação de desempenho no cenário A.
   B. Maior degradação de desempenho no cenário B.
   C. A degradação de desempenho será semelhante em ambos os cenários.
   D. Não haverá degradação de desempenho em nenhum dos casos.

Versão: v0 (deve corresponder ao algarismo menos significativo do número de aluno)

3. [3,3v] Pretende-se implementar uma função *ht_change_key*, que muda a chave de um elemento já existente na tabela de dispersão. Como, após a alteração da chave, o valor de dispersão (*hash value*) associado ao elemento pode ser diferente, pode ser necessário transferir o elemento para outra lista.

Considere a seguinte implementação da nova função, que ainda não inclui qualquer sincronização.

```
int  ht_change_key(key_t oldkey, key_t newkey) {
  element_t *e;

  //Check if new key is not already taken
  if (ht_lookup(newkey) != NULL)
    return -1;

  //Get the element whose key is to be changed
  e = ht_lookup(oldkey);
  if (e == NULL)
    return -1;

  //Change the key (this may require moving element
  //to a different list)
  if (hash(oldkey) == hash(newkey)) {
    e->key = newkey;
    return 0;
  }
  else {
    ht_delete(oldkey);
    e->key = newkey;
    ht_insert(e);
  }
  return 0;
}
```

Complete a implementação com a sincronização necessária, assumindo que a função pode ser chamada concorrentemente. Na sua resposta, apenas tem de indicar que linhas adicionaria ou modificaria ao programa original.

Na sua solução: i) tenha especial cuidado para prevenir situações de interblocagem; ii) deve manter a estratégia de *mutexes* finos (tal como na solução apresentada).

```
void acquireMutexes(int m1, int m2) {
  if (m1 == m2)
    pthread_mutex_lock(&mutex[m1]);
  else {
    pthread_mutex_lock(&mutex[min(m1, m2)]);
    pthread_mutex_lock(&mutex[max(m1, m2)]);
  }
}

void releaseMutexes(int m1, m2) {
  pthread_mutex_unlock(&mutex[m1]);
  if (m1 != m2)
    pthread_mutex_unlock(&mutex[m2]);
}
```

```
int  ht_change_key(key_t oldkey, key_t newkey) {

  acquireMutexes(hash(oldkey), hash(newkey));

// Modificações no resto do código:
//1. Substituir chamadas a ht_lookup, ht_insert, ht_delete por
//  chamadas a variantes ht_lookup_nosync, ht_insert_nosync,
//  ht_delete_nosync (respetivamente), com a mesma lógica das
//  funções originais mas sem a sincronização
//2. Antes de qualquer return, colocar chamada a
//  releaseMutexes(hash(oldkey), hash(newkey));

}
```

Versão: v0 (deve corresponder ao algarismo menos significativo do número de aluno)

4. [1,6v] Usando o programa com muitas tarefas, verificou-se que o desempenho era medíocre pois as tarefas passavam muito tempo bloqueadas nas funções *lock.* No entanto, uma análise do comportamento das tarefas revelou que a grande maioria das chamadas são à função *ht_lookup*. O que modificaria na implementação proposta para otimizar o desempenho?
   A. Substituir o vetor de mutexes por um vetor de trincos de leitura-escrita. Na função ht_lookup trancar para leitura. Nas funções ht_insert e ht_delete trancar para escrita.
   B. Adicionar um novo vetor de trincos de leitura-escrita. Na função ht_lookup, trancar para leitura o respetivo trinco de leitura-escrita. Não modificar as restantes funções.
   C. Adicionar um novo vetor de mutexes. A função ht_lookup passa fechar os mutexes do novo vetor (em vez do vetor *mutexes[N]*). Não modificar as restantes funções.
   D. Substituir mutexes por variáveis de condição.

5. [3,3v] Pretende-se que cada lista da tabela de dispersão não tenha mais que M elementos. Sempre que uma tarefa chame a função *ht_insert* e a lista respetiva já tenha M elementos, a função deve esperar até que algum elemento dessa lista seja removido (por outra tarefa que chame *ht_remove*).
   Apresente as modificações que faria à implementação da tabela de dispersão para suportar esta alteração. Como ponto de partida, assuma a implementação original apresentada neste enunciado (ou seja, ignore as modificações das alíneas anteriores).

```
pthread_cond_t podeinserir[N];
//Por simplicidade, omitimos a inicialização da variável de
condição

//No código seguinte, assumimos que existe uma função list_count,
//que retorna o número de elementos na lista; alternativamente,
//podia ser declarado e mantido um contador (incrementado na
//inserção, decrementado na remoção)
void ht_insert(element_t *e) {
  [...]
  pthread_mutex_lock(&mutex[h]);
  while (list_count(hashtable[h]) == M)
    pthread_cond_wait(&podeinserir[h], &mutex[h]);
  [...]
}

void ht_delete(key_t k) {
  [...]
  pthread_cond_signal(&podeinserir[h]);
  pthread_mutex_unlock(&mutex[h]);
}
```