# Sistemas Operativos, Teste 1, 28/outubro/2017

IST - LEIC-A/ LEIC-T/ LETI - 2017-2018

Soluções

| **Folha de Respostas (1/4)** |
|---|
| **Número:**<br>**Nome:** |

§

Tarefas e Troca de Mensagens

| Pergunta 1 | |
|---|---|

```
/*-------------------------------------------------------------------
| Defines
-------------------------------------------------------------------*/
#define BUFFSZ 256

typedef struct {
  int    id;
  char  *token;

} argsSimular_t;

/*-------------------------------------------------------------------
| Function: apply_token
-------------------------------------------------------------------*/
void *apply_token(void *a) {
   argsSimular_t *arg   = (argsSimular_t *) a;
   int            myid     = arg->id;
   char           buffer[BUFFSZ];
   int            occurrences = 0;
   char*          token = arg->token;

   while (1) {
     receberMensagem(arg->id-1, arg->id, buffer, BUFFSZ);
     occurrences = toupper_token (buffer, token);
     if (ocurrences >0)
         enviarMensagem(arg->id, arg->id+1, buffer, BUFFSZ);
   }
   return 0;
}


/*-------------------------------------------------------------------
| Function: printer
-------------------------------------------------------------------*/
void *printer(void *a) {
   argsSimular_t *arg   = (argsSimular_t *) a;
   int             myid = arg->id;
   char            buffer[BUFFSZ];

   while (1) {
     receberMensagem(arg->id-1, arg->id, buffer, BUFFSZ);
     printf ("%s\n",buffer);
   }
   return 0;
}
```

**Folha de Respostas (2/4)**

**Número:**
**Nome:**

§

Tarefas e Troca de Mensagens

| Pergunta 1 | (continuação) |
|---|---|

```
/*-------------------------------------------------------------------
| Function: main
-------------------------------------------------------------------*/


int main (int argc, char** argv) {
  argsSimular_t     slave_args[3];
  pthread_t         slaves[3];
  char              *token1 = argv[1];
  char              *token2 = argv[2];
  char              buffer[BUFFSZ];

  if (argc < 3) {
    printf("pipeline token1 token2\n");
    return 1;
  }

  /* Inicializa biblioteca de troca de mensagens
  (capacidade do canal=10, número de tarefas comunicantes=4) */
  inicializarMPlib(10,4);

  slave_args[0].id = 1;
  slave_args[0].token = token1;
  pthread_create(&slaves[0], NULL, apply_token, &slave_args[0]);

  slave_args[1].id = 2;
  slave_args[1].token = token2;
  pthread_create(&slaves[1], NULL, apply_token, &slave_args[1]);

  slave_args[2].id = 3;
  pthread_create(&slaves[2], NULL, printer, &slave_args[2]);


  // le linhas do stdin
  while (fgets (buffer, BUFFSZ, stdin)) {
      enviarMensagem(0, 1, buffer, BUFFSZ);
  }
}
```

**Folha de Respostas (3/4)**

**Número:**
**Nome:**

§

Sincronização com Mutexes e Variáveis de Condição

| | |
|---|---|
| Pergunta 2 | Cada tarefa, antes de obter o token, bloqueia-se caso `c==0`.<br>Para além disso, antes de aceder ao token, uma tarefa que não tenha sido bloqueada decrementa a variável $c$.<br>Percebe-se que o objetivo do programador seria limitar o número de tarefas simultaneamente com o token.<br>Em particular, com o valor inicial de `c=5`, teríamos no máximo 5 tarefas com o token.<br>Infelizmente o código tem um bug, pelo que este objetivo não é conseguido.<br>Devido ao bug, não há limite ao número de tarefas que conseguem aceder simultaneamente ao token (ver alínea seguinte). |
| Pergunta 3 | O problema resulta da variável $c$ ser decrementada fora da exclusão mútua.<br>Considere que `c==1`.<br>A tarefa $t_1$ fecha o trinco (mutex) sem se bloquear, lê o valor de $c$, liberta o trinco e perde o processador antes de conseguir decrementar a variável $c$.<br>A tarefa $t_2$ ganha o processador e ainda vê a variável `c==1`.<br>Neste caso, $t_2$ também consegue fechar o trinco sem se bloquear (apesar de $t_1$ já ter conseguido o acesso ao token). |
| Pergunta 4 | A variável $c$ deveria ser decrementanda *antes* de se libertar o trinco |
| Pergunta 5 | Numa espera ativa, uma tarefa $t_1$ está num ciclo à espera que uma variável mude de valor.<br>Existindo apenas um processador, enquando $t_1$ não perder o processador, o valor dessa variável não irá mudar (é necessário que outro código se execute para mudar o valor).<br>Isso nunca acontecerá antes de ocorrer uma interrupção (por exemplo, quando passado um dado *quantum* de tempo o núcleo for chamado e realizar uma troca de contexto).<br>Isto leva a um desperdício de recursos.<br>Literalmente, $t_1$ está a trabalhar para aquecer (o processador)! |

**Folha de Respostas (4/4)**

**Número:**
**Nome:**

§

Sincronização com Semáforos

| Pergunta 6 | |
|---|---|

```
#define TIPOA 0
#define TIPOB 1

#define outro(x) x? 0: 1

// esperavez garante que não passam mais do que 1 de um dado tipo
Semaforo esperavez[2] = {1, 0};

// conta quantos já passaram pelo recurso (de um dado tipo)
int       nporturno = 2;


void pedeAcesso (int tipo) {

    // espero que o meu tipo possa aceder
    esperar (esperavez[tipo]);
}

void libertaAcesso (int tipo) {

    nporturno--;
    if (nporturno == 1) {
       // o próximo ainda é do meu tipo
       assinalar (esperavez[tipo]);
    }
    else {
      // já passaram 2 do meu tipo
      // vou deixar passar 2 do outro tipo
      nporturno = 2;
      assinalar (esperavez[outro(tipo)]);
    }
}
```