ISEP – LETI – Sistemas Operativos e Computacionais

Ficha 5 – Sincronização de tarefas: *mutex* e semáforo

1) Analise o seguinte extrato de um programa:

```
int main() {
  int *v = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
                MAP_SHARED | MAP_ANONYMOUS, -1, 0);

  //semáforo com nome
  sem_t *psem = sem_open("/sem1", O_CREAT | O_RDWR, 0600, 0);

  v[0] = 0;
  int r, n = 0;
  r = fork();
  ++n;
  ++v[0];
  if(r == 0) {
    sem_wait(psem);
    printf("%d: *v = %d, n = %d\n", getpid(), v[0], n);
    return(0);
  }
  sleep(5);
  printf("%d: *v = %d, n = %d\n", getpid(), v[0], n);
  sem_post(psem);
  return(0);
}
```

Apresente a sequência de impressões produzidas por este programa. Assuma que não existem interferências de outros processos no sistema, que o identificador do processo inicial é 2000 e que o(s) novo(s) processo(s) toma(m) o(s) valor(es) seguinte(s). Apresente um diagrama temporal representativo da execução do programa e justifique sucintamente. A sequência de impressões deve ser apresentada de forma destacada.

Note que a *flag* `MAP_ANONYMOUS` permite que a função `mmap` crie um bloco de memória partilhada sem ser necessário recorrer à função `shm_open`.

2) Extraia o conteúdo do ficheiro `ficha-sinc-ficheiros.zip` para o seu diretório de trabalho. Analise o programa contido em `ex2.c`. A função `myprint` simula a escrita num dispositivo "lento". Esta função foi implementada de forma a realçar o risco de conflitos no acesso concorrente a recursos partilhados; no entanto, mesmo com as habituais operações de escrita, este tipo de problemas também poderá ser verificado, embora de forma menos evidente.

Pode utilizar o comando `make` para criar o executável **`ex2`**.

a) Execute o programa. Justifique o comportamento observado (*strings* "misturadas") com base no descrito acima.

b) No caso deste programa, o recurso partilhado é o ecrã (ambos os processos concorrem entre si para poderem imprimir no ecrã). Identifique os pontos do programa em que o acesso pode ocorrer em simultâneo.

c) Utilize o mecanismo de semáforos com nome para garantir que a impressão de cada processo é enviada para o ecrã sem ser interrompida pelas impressões do outro processo.

Observação: tenha atenção ao facto que o semáforo com nome mantém o seu estado mesmo após a terminação dos processos que o utilizam. Para repor o valor inicial do semáforo, terá que o apagar (`sem_unlink`).

3) Considere o seguinte extrato de código do ficheiro `ex3.c`:

```
int main()
{
  char *buf;
  pthread_t tid1, tid2;

  buf = (char *) malloc(256);
  sprintf(buf, "%d: Ola ------------------------------------------------------------\n",getpid());
  /*
  Irá ser criada uma nova thread e executada a função mythread(buf) nessa thread.
  A variável buf contém    o endereço do bloco de memória onde é guardada a string.
  */
  pthread_create(&tid1, NULL, mythread, buf);



  buf = (char *) malloc(256);
  sprintf(buf, "%d: Ole ++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n",getpid());
  pthread_create(&tid2, NULL, mythread, buf);

  //...

  pthread_join(tid1, NULL);

  return(0);
}


void *mythread(void *arg){
  char *str1 = (char *) arg;
  while(1) {
    sleep(1);

    myprint(str1);
  }
}
```

a) Utilize um *mutex* para garantir que a impressão de cada *thread* é enviada para o ecrã sem ser interrompida pelas impressões da outra *thread*. Não deverá alterar a função `myprint`.

b) Implemente o solicitado no ponto anterior, recorrendo a um semáforo sem nome em vez do *mutex* (ver exemplo dos slides das aulas teóricas).

4) Crie novas versões dos programas que implementou no Exercício 1 (memória partilhada) da Ficha sobre mecanismos de comunicação entre processos de modo a garantir que o processo "leitor" apenas imprime a *string* contida na memória partilhada quando um processo "escritor" escreve uma nova *string* nessa mesma memória partilhada. Implemente o comportamento desejado recorrendo a um semáforo com nome.

5) Considere a API de filas de mensagem da norma POSIX (`man 7 mq_overview, man mq_send, man mq_receive`).

Pretende-se implementar uma biblioteca de fila de mensagens para programas *multithread*, de acordo com as declarações e comentários apresentados abaixo (ficheiro `mymq.h`).

```
typedef struct
{
  int slot_size;           //maximum slot size. Set by mymq_init.
  int number_of_slots;     //number of slots. Set by mymq_init.
  void *slots;             //malloc(slot_size*number_of_slots).
  int *msg_size;           //indicates the actual size of the message stored
                           //on each slot
  int oldest_slot;         //index of the oldest element in the queue
  sem_t sem_msgs_in_queue; //send() increments this, receive() waits on this
  sem_t sem_free_slots;    //receive() increments this, send() waits on this
  pthread_mutex_t access;  //the accesses to the message queue
                           //must be mutually exclusive
}
mymq_t;


//This function must be called to initialize the queue. Returns -1 on error.
int mymq_init(mymq_t *mq, int slot_size, int number_of_slots);

void mymq_send(mymq_t *mq, void *data, int size);

//Returns size of received message  (in bytes)
int mymq_receive(mymq_t *mq, void *data, int size);

//Releases all resources used by the queue
int mymq_destroy(mymq_t *mq);
```

A fila de mensagens só poderá ser usada para troca de mensagens entre *threads* do mesmo processo e todas as mensagens deverão ter a mesma prioridade ("*first in, first out*"). De resto, pretende-se que as funções `mymq_send` e `mymq_receive` tenham um comportamento análogo ao comportamento *default* das funções `mq_send` e `mq_receive` da norma POSIX. O ficheiro `mymq_test.c` apresenta um exemplo de utilização das funções a implementar.

Sugestões de implementação:

- Os bloqueios de fila cheia, no caso da função `mymq_send`, e de fila vazia, no caso da função `mymq_receive`, deverão ser geridos com os semáforos declarados na estrutura `mymq_t` (campos `sem_free_slots` e `sem_msgs_in_queue` respetivamente).
- O campo `slots` deverá apontar para um bloco de memória a ser usado como um *buffer* circular, onde serão armazenadas as mensagens. Um *buffer* circular é um vetor em que os dados são armazenados e consumidos de forma circular, isto é, quando se atinge o último elemento do vetor (índice `number_of_slots-1`, neste caso), o próximo elemento a considerar será o primeiro (índice 0). Nesse sentido:
    - O campo `oldest_slot` deverá ser incrementado (usando aritmética "modular") sempre que uma mensagem é lida da fila:

      ```
      oldest_slot = (oldest_slot + 1) % number_of_slots.
      ```

    - Novas mensagens deverão ser guardadas na posição indicada pela seguinte expressão:

      ```
      (oldest_slot+número_de_mensagens_na_fila) % number_of_slots.
      ```

      O número de mensagens na fila é indicado pelo semáforo `sem_msgs_in_queue` (usar a função `sem_getvalue` para consultar o valor do semáforo).
- O campo `msg_size` deverá também armazenar o endereço base de um vetor. Neste caso, trata-se de um vetor de inteiros que deverá ter o mesmo número de elementos do vetor apontado por `slots` e deverá ser indexado da mesma forma. A finalidade deste vetor é indicar, para cada *slot*, o tamanho da mensagem armazenada. Deverá ser iniciado a zero.

A função `mymq_init` deverá ser usada para iniciar corretamente cada um dos campos da variável do tipo `mymq_t` apontada pelo parâmetro `mq`.

De seguida, apresenta-se um cenário de utilização da fila, com resultados esperados:

```
mymq_t mq;
mymq_init(&mq, 10, 3); //3 slots of 10 bytes
mymq_send(&mq, "Test", 4);
```

Slots	TestXXXXXX	XXXXXXXXXX	XXXXXXXXXX
msg_size	4	X	X
oldest_slot	0		
sem_msgs_in_queue	1		
sem_free_slots	2		

X – valor/byte indefinido

```
mymq_send(&mq, "Test2", 5);
```

slots	TestXXXXXX	Test2XXXXX	XXXXXXXXXX
msg_size	4	5	X
oldest_slot	0		
sem_msgs_in_queue	2		
sem_free_slots	1		

```
mymq_send(&mq, "Test345", 7);
```

slots	TestXXXXXX	Test2XXXXX	Test345XXX
msg_size	4	5	7
oldest_slot	0		
sem_msgs_in_queue	3		
sem_free_slots	0		

```
char buffer[10];
mymq_receive(&mq, buffer, 10);
```

slots	TestXXXXXX	Test2XXXXX	Test345XXX
msg_size	4	5	7
oldest_slot	1		
sem_msgs_in_queue	2		
sem_free_slots	1		

```
mymq_send(&mq, "TestFinal", 9);
```

slots	TestFinalX	Test2XXXXX	Test345XXX
msg_size	9	5	7
oldest_slot	1		
sem_msgs_in_queue	3		
sem_free_slots	0		

a) Implemente as 4 funções descritas.
b) Teste a biblioteca desenvolvida com o programa do ficheiro `mymq_test.c`.

Histórico

- 2023-03-15 – Criação do documento. Exercício 1, 2, 3 e 5 são baseados em exercícios das fichas de SISTC da LEEC - Jorge Estrela da Silva (jes@isep.ipp.pt)