



## ***IPC— Threads e Monitores***

### ***Biblioteca pthread em Unix***

#### **Objectivo**

Estudo da implementação de *threads* em Unix. Utilização das funções da biblioteca *pthread* `pthread_create`, `pthread_join`, `pthread_exit`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_cond_wait`, `pthread_cond_signal`, `pthread_once` e `pthread_self`.

#### **Guião**

1. *Condições de corrida no acesso a uma região partilhada*
  - a) Entre no directório `incrementer` e analise o código do módulo descrito nos ficheiros `incMod.h` e `incMod.c` que definem uma variável interna e três primitivas de acesso (`vSet`, `vGet` e `vInc`) para a sua manipulação.
  - b) O programa `incrementer.c` ilustra os passos de lançamento de *threads* e de espera pela sua terminação usando o módulo anterior como instrumento de interação. Procure determinar o que resulta da sua execução com os valores por omissão. Consulte no manual *on-line* a descrição das chamadas ao sistema `pthread_create`, `pthread_join` e `pthread_exit`.
  - c) Crie o ficheiro executável `incrementer` (*make incrementer*), execute o programa com os valores por omissão e confirme as suas deduções.
  - d) Assuma agora que, em vez de um *thread*, serão lançados vários. Que diferenças vão ocorrer? Para entender bem o problema que é colocado, procure responder cuidadosamente às questões seguintes
    - i) Em que consiste a região partilhada?
    - ii) Ponha em destaque o código da sua manipulação, aquilo que se designa comumente de *região crítica*.
    - iii) Construa um diagrama ilustrativo do ciclo de vida dos *threads* que identifique com clareza os diferentes componentes da região crítica.
    - iv) Mostre sobre o diagrama anterior como é que vão ocorrer *condições de corrida* na execução do programa se o número de *threads* lançado for superior a um. Admita nas suas considerações que o programa é executado num monoprocessador (num processador *multicore* a situação é essencialmente semelhante) e que há apenas dois *threads*.
  - v) *Condições de corrida* conduzem habitualmente a inconsistência de informação.

Assuma o lançamento de  $N$  *threads* e que se repete o ciclo de incremento da variável 1000 vezes. Entre que limites variará o *valor final* que é impresso? Justifique detalhadamente a sua resposta.

- vi) Considere finalmente que se varia o valor dos atrasos 1 e 2 da primitiva `vInc` alterando o parâmetro dos atrasos de `BIG` para `SMALL`. Admitindo as quatro combinações possíveis, que valores são de esperar para o *valor final* em cada um dos casos?
- e) Usando as opções disponíveis, nomeadamente o lançamento de um número variável de *threads*, e variando o valor dos atrasos 1 e 2, execute o programa diversas vezes (repita pelo menos cinco vezes a execução em cada caso). Interprete os resultados obtidos.

## 2. Imposição de exclusão mútua no acesso a uma região crítica

- a) Um *mutex* é um dispositivo que implementa a exclusão mútua no acesso a uma região crítica. A biblioteca *pthread* define o tipo de dados `pthread_mutex_t` e as funções `pthread_mutex_lock` e `pthread_mutex_unlock` para a sua manipulação. Consulte no manual *on-line* a descrição destas funções.
- b) Ao ser garantida a exclusão mútua na execução das primitivas de acesso, um módulo transforma-se num *monitor*. O ficheiro `incModSafe.c` implementa o módulo `incMod` como um monitor. Analise o seu código.
- c) Crie o ficheiro executável `incrementerSafe` (*make incrementerSafe*) e execute o programa diversas vezes nas mesmas condições usadas anteriormente. Compare os resultados agora obtidos com os anteriores.

## 3. Sincronização de operações num monitor e inicialização da estrutura de dados interna

- a) Uma *variável de condição* é um dispositivo de sincronização que permite suspender a execução de um *thread* no interior do monitor até à satisfação de uma dada condição. A biblioteca *pthread* define o tipo de dados `pthread_cond_t` e as funções `pthread_cond_wait` e `pthread_cond_signal` para a sua manipulação. Consulte no manual *on-line* a descrição destas funções.
- b) Entre no directório `prodcon` e analise o código do monitor descrito nos ficheiros `fifo.h` e `fifo.c` que definem uma memória de tipo FIFO e as duas primitivas base para a sua manipulação. Procure responder às questões seguintes
  - i) Como é definido o tamanho do FIFO?
  - ii) Porque é que é necessário sincronizar as operações de inserção e de retirada de valores num ambiente de invocação concorrente? Que condições são usadas para a sua especificação?
  - iii) Construa o diagrama de estados ilustrativo da operação da memória.
  - iv) Como é que a estrutura de dados, que implementa a memória de armazenamento, é inicializada? Como é que é garantido que a inicialização ocorre uma e uma única vez? Consulte no manual *on-line* a descrição da função `pthread_once`.
- c) O programa `prodcon.c` ilustra um problema clássico em programação concorrente, o chamado *problema dos produtores-consumidores*, em que um grupo de *threads*, ditos

*produtores*, produzem informação e comunicam-na a um segundo grupo de *threads*, ditos *consumidores*, que a consomem de algum modo. A memória de tipo FIFO é usada neste caso como dispositivo de comunicação. Construa o diagrama de interacção que lhe está subjacente e indique que tipo de impressão vai ser produzida quando o programa for executado.

- d) Crie o ficheiro executável `prodcon` (*make prodcon*), execute o programa e confirme as suas deduções.