Exemplo

Uso de Variáveis condicionais

Aviso de uma *thread* a outra que algo foi ocorreu

Este exemplo ilustra o uso de variáveis condicionais para uma *thread* avisar outra que algo aconteceu. Esse algo corresponde a uma etapa de trabalho feita pela *thread* sem que esta termine. Isto significa que a *thread* interessada em perceber que esse algo ocorreu não pode usar a estratégia de *pthread_join* dado que a *thread* que produz o acontecimento (faz esse algo) permanece em execução, eventualmente produzindo diversos algos

O exemplo em questão constrói um *array* enorme de inteiros inicializado a zeros e coloca em posições aleatórias alguns valores 123. O programa vai depois procurar esses valores até encontrar uma determinada quantidade de valores 123. O exemplo simboliza o típico problema de encontrar valores de interesse num espaço de procura muito grande. A estratégia passa, inevitavelmente, por dividir o espaço de procura – o *array* – por várias *threads* e cada uma procura apenas no seu espaço. Como trabalham em paralelo (até esgotar o número de cores disponíveis no processador será mesmo paralelo verdadeiro), tem-se um ganho de performance real e trata-se de um exemplo de uso de *threads* para ganhos de performance. Haverá então várias *threads* (mesma função) que efetuam a pesquisa, e uma outra *thread* que orquestra (organiza) tudo, dando ordem para iniciar a pesquisa e aguardado pelos resultados.

Assumindo que há mais valores 123 do que aqueles que o programa pretende encontrar, interessa que a *thread* que está a orquestrar a procura seja avisada de cada valor encontrado à medida que vão sendo encontrados de forma a poder dar ordem de término às *threads* que procuram assim que o número desejado de valores 123 for atingido, evitando procurar mais sem necessidade. Trata-se assim do cenário em que as *threads* que procuram produzem algo sem terminar de imediato.

O aviso de mais um valor encontrado é feito através de uma variável condicional. A *thread* que orquestra (a executar a função *main*), após colocar as *threads* que pesquisam a trabalhar, ficará a aguiar por avisos (*pthread_cond_wait*) na variável condicional, e quando atinge o número de avisos pretendido, dá indicação às restantes *threads* para encerrarem pela via habitual (variável “continua”). As *thread* que pesquisam, sempre que

encontram um valor 123 efetuam um aviso na variável condicional (*pthread_cond_signal*), prosseguindo de imediato a sua pesquisa.

A dimensão do *array*, o número de valores 123 a semear inicialmente nesse *array*, o número desejado de valores a encontrar, e o número de *threads* de pesquisa são configurados pelo utilizador em linha de comandos. O programa tem um mecanismo para atrasar artificialmente o seu funcionamento, que pode ou não ser usado (a invocação da função que implementa esse atraso está em comentários).

O programa tem duas situações possíveis de *deadlock* cuja resolução se deixa como desafio (foram debatidas nas aulas):

- Tentar procurar mais valores 123 do que aqueles que existem no *array*. Nesta situação, as *threads* terminam naturalmente permanecendo a *thread* que orquestra à espera de mais avisos indefinidamente.
- Existência de valores 123 em posições muito próximas do início do espaço de procura de cada *threads*. Isso fará com que sejam disparados múltiplos avisos (*cond signals*) antes que a função que orquestra esteja preparada para os receber, por estar ainda a criar as *threads* de pesquisa restantes. Os múltiplos cond signals acumulam-se, em que o último mascara os anteriores, fazendo com que, no final, fiquem a faltar alguns avisos. Se não houver valores 123 a mais em número suficiente, fica-se numa situação semelhante à do ponto anterior.

Estes casos são duplamente interessantes: garantir que se consegue ver, por análise do código, a existência destas situações, e depois, a capacidade de as remediar.

O código do programa é apresentado a seguir. Segue uma estratégia alinhada com os restantes exemplos de *threads* já apresentados e não necessita de explicações adicionais. O código de deteção de erros nas chamadas ao API foi reduzido de forma a não desfocar o código do ponto que se pretende ilustrar.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

typedef struct {
    int *arr;
    long inicio;
    long fim;
    int * continua;
    // -- var conds
    pthread_mutex_t *mutex;
    pthread_cond_t *cond;
    // -- id da thread
    pthread_t tid;
} DadosThread;
```

```
void simulaMaisLento() {
    int a=0, b=0;
    for (int i=1; i<=100; ++i)
       for (int j=1; j<=1000; ++j)
          a += j*i % (j+i);  // contas sem significado para perder algum tempo
    b = a;
}

void *funcaoThread(void *arg) {
    DadosThread *dados = (DadosThread*)arg;

    for (long i = dados->inicio; i < dados->fim; ++i) {

        // Se a main já mandou parar, sai do ciclo
        if (!*(dados->continua))   // a continua é partilhada. Logo, devia ser
            break;                 // protegida (race condition) (TPC)

        // simula não ser tão rápico. Descomentar para usar
        // simulaMaisLento();

        // se encontrou um 123, avisa a main (que o irá contabilizar)
        if (dados->arr[i] == 123) {
            pthread_mutex_lock(dados->mutex);

                // secção crítica A
                // A thread podia incrementar o contador aqui
                // e mandar parar as restantes, mas é mais
                // natural ser a função main a gerir isso

            pthread_cond_signal(dados->cond);

                // secção crítica B

            pthread_mutex_unlock(dados->mutex);
        }
    }

    return NULL;
}

long tempoMS() {    // para para medir o tempo inicial e final (receita)
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec*1000 + ts.tv_nsec/1000000;
}

int main(int argc, char *argv[]) {
    if (argc != 5) {
        printf("Uso: %s tam-array num-valores-123-a-semear"
               "Quantidade-a-procurar num-threads\n", argv[0]);
        return 1;
    }
```

```
long int tamArray   = atol(argv[1]);
long int numSemear  = atol(argv[2]);
int totalPretendido = atoi(argv[3]);
int numThreads      = atoi(argv[4]);

int contador = 0;
int continuaGeral = 1;  // uma única Continua para todas as threads

long tamanhoParte = tamArray / numThreads;  // parte do array para
                                            // cada thread
setbuf(stdout, NULL);

int * array = malloc(sizeof(int) * tamArray);
if (!array) {
    perror("\nerro no malloc");
    exit(1);
}

for (long i = 0; i < tamArray; ++i)
    array[i] = 0;

// semear valores 123 em posições aleatórias
// rand() - receita habitual da biblioteca std C
srand(time(NULL));
for (long i = 0; i < numSemear; ++i) {
    long pos = rand() % tamArray;
    array[pos] = 123;
}

DadosThread tdados[numThreads];

// mecanismos de sincronização: var condicional (+mutex)

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

long t0 = tempoMS();  // regista tempo inicial

// cria threads
for (int i = 0; i < numThreads; ++i) {
    tdados[i].arr = array;
    tdados[i].inicio = i * tamanhoParte;
    tdados[i].fim = (i == numThreads-1 ? tamArray :
                                         (i+1)*tamanhoParte);
    tdados[i].continua  = & continuaGeral;   // aponta para a
                                             // continua comum
    tdados[i].mutex = & mutex;  // mesmo mutex para todas, obviamente
    tdados[i].cond = & cond;    // idem variável condicional

    pthread_create(& tdados[i].tid, NULL, funcaoThread, & tdados[i]);
}
```

```
    // main aguarda notificações das threads

    pthread_mutex_lock(&mutex);

    while (continuaGeral) {

            // tem o mutex - isto é uma secção crítica
            // é seguro aqui aceder a dados partilhados

        // espera por sinal das threads
        // Importante: este wai vai libertar o mutex
        pthread_cond_wait(&cond, &mutex);
        // avançou => readquiriu o mutex

            // outra secção crítica (está na posse do mutex)
            //   mas é ua outra secção crítica - indep. da anterior
            // é seguro aqui aceder a dados partilhados

        ++contador;   // mais um 123 encontrado
        if (contador >= totalPretendido)
            continuaGeral = 0;  // encerrar todas

    }

    pthread_mutex_unlock(&mutex);


    // esperar por todas as threads
    for (int i = 0; i < numThreads; ++i)
        pthread_join(tdados[i].tid, NULL);

    long t1 = tempoMS();

    printf("\nEncontrados = %d", contador);
    printf("\nTempo = %ld ms\n", t1 - t0);

    free(array);
    return 0;
}
```

Experiências que se podem fazer com este exemplo

- Executar e perceber o papel das variáveis condicionais no aviso das *threads* que procuram os valores à *thread* que executa a *main*.
- Ver o funcionamento com diferente configurações e observar a ocorrência de *deadlocks*.
- Resolver a possibilidade de *deadlock* usando as várias abordagens referidas nas aulas.