Resumo Programação Concorrente

Capítulo 1 - Introdução à computação concorrente

A programação concorrente está relacionada com a atividade de construir programas de computador que incluem linhas de controle distintas, as quais podem executar simultaneamente. Dessa forma, um programa dito concorrente se diferencia de um programa dito sequencial por conter mais de um fluxo de execução independente.

Para escrever um programa concorrente é necessário definir quais fluxos de execução criar e de que forma eles deverão interagir dentro da aplicação. As decisões devem levar em conta as especificidades da aplicação e o hardware disponível.

A biblioteca utilizada será a `pthread.h` na linguagem C

Primeiro Programa Concorrente (Hello World!)

Antes de seguirmos vamos ver as primeira funções que usaremos:

Criação de Threads

Retorna 0 em caso de sucesso

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void *
(*start_routine)(void*), void *arg);
```

Exemplo de uso:

```
//identificadores das threads no sistema
pthread_t tids[nthreads];
//cria as threads e atribui a tarefa de cada thread
for( long int i=0; i<nthreads; i++){
    if(pthread_create(&tids[i], NULL, task, (void*) i)){
        printf("Erro pthread_create %ld\n",i);
        return 2;
    }
}
```

Encerramento de Threads

A função encerra o thread de chamada e cria o valor *retval* disponível para qualquer junção bem-sucedida

```
void pthread exit (void *retval);
```

Espera pelo encerramento das threads

A função suspende a execução da thread chamada até que a execução das threads termine, em caso de sucesso ela retornará 0.

```
int pthread join (pthread t thread, void **retval);
```

Exemplo de uso:

```
for(short int i=0; i<nthreads;i++ ){
    if(pthread_join(tids[i],NULL)){
        printf("Error!");
    }
}
```

Tarefa a ser executada pelas threads

A para definir o que as threads criadas terão que fazer, faremos uma função que será chamada em `pthread_join`

```
void* tarefa(void* arg){
    /* Código a ser executado por cada thread*/
    pthread_exit(NULL);
}
```

helloWorld.c

Normalmente em um programa sequencial em C temos a seguinte estrutura

```
#include <stdio.h>

int main(){
    /* código */
return 0;
}
```

Para um programa concorrente, manteremos essa estrutura base e incorporamos as funções acima.

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

```c
#include <stdlib.h>

// funcao executada pelas threads
void* task(void* arg){
    long int id = (long int) arg;
    printf("Hello World! %ld\n", id);
    pthread_exit(NULL);
}

// Funcao principal
int main(int argc, char *argv[])
{

//qtde de threads que serao criadas (recebida na linha de comando
    short int nthreads;

//verifica se o argumento 'qtde de threads' foi passado e armazena seu valor
    if(argc < 2 ){
        printf("Error! digite %s <n de threads>", argv[0]);
        return 1;
    }

    nthreads = atoi(argv[1]); //capturando o num de threads inserida pelo prompt

//identificadores das threads no sistema
    pthread_t tids[nthreads];
     //cria as threads e atribui a tarefa de cada thread
    for( long int i=0; i<nthreads; i++){
        if(pthread_create(&tids[i], NULL, task, (void*) i)){
            printf("Erro pthread_create %ld\n",i);
            return 2;
        }
    }

// Espera todas as thread terminarem
    for(short int i=0; i<nthreads;i++ ){
        if(pthread_join(tids[i],NULL)){
            printf("Error!");
        }
    }

    //log da funcao main
    printf("FIM!");
    return 0;
}
```

Passando mais de um argumento para as threads

Quando queremos passar mais argumentos para um thread nós utilizamos estrutura (`struct`)
E utilizamos o parâmetro `(void* arg)` em `pthread_create` para enviar todas as informações necessárias para a thread

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

//cria a estrutura de dados para armazenar os argumentos da thread
typedef struct {
   int idThread, nThreads;
} t_Args;

//funcao executada pelas threads
void *PrintHello (void *arg) {
  //typecarting do argumento
  t_Args args = *(t_Args*) arg;

  //log da thread
  printf("Sou a thread %d de %d threads\n", args.idThread, args.nThreads);

  free(arg); //libera a memoria que foi alocada na main

  pthread_exit(NULL);
}

//funcao principal do programa
int main(int argc, char* argv[]) {
  t_Args *args; //receberá os argumentos para a thread
  int nthreads; //qtde de threads que serao criadas (recebida na linha de comando)

  //verifica se o argumento 'qtde de threads' foi passado e armazena seu valor
  if(argc<2) {
      printf("--ERRO: informe a qtde de threads <%s> <nthreads>\n", argv[0]);
      return 1;
  }
  nthreads = atoi(argv[1]);

  //identificadores das threads no sistema
  pthread_t tid_sistema[nthreads];

  //cria as threads
  for(int i=1; i<=nthreads; i++) {
    printf("--Aloca e preenche argumentos para thread %d\n", i);
    args = malloc(sizeof(t_Args));
    if (args == NULL) {
      printf("--ERRO: malloc()\n");
      return 2;
    }
    args->idThread = i;
    args->nThreads = nthreads;

    //printf("--Cria a thread %d\n", i);
    if (pthread_create(&tid_sistema[i-1], NULL, PrintHello, (void*) args)) {
      printf("--ERRO: pthread_create() da thread %d\n", i);
    }
  }
```

```
    //log da função principal
    printf("--Thread principal terminou\n");

    pthread_exit(NULL);
}
```

Capítulo 2 - Construindo algoritmos concorrentes

Tipos de problemas concorrentes

Para um conjunto de dados D

$$D = d_1 \oplus d_2 \oplus \cdots \oplus d_k = \sum_{i=1}^{k} d_i$$

Paralelismo de Dados

Dizemos que o problema P_D é um problema com paralelismo de dados se D é composto de elementos de dados e é possível aplicar uma função $f(\ldots)$ para todo o domínio:

$$f(D) = f(d_1) \oplus f(d_2) \oplus \cdots \oplus f(d_k) = \sum_{i=1}^{k} f(d_i)$$

Paralelismo de Tarefas

Por outro lado, se D é composto por funções e a solução do problema consiste em aplicar cada função sobre um fluxo de dados comum S, dizemos que o problema P_D é um problema com paralelismo de tarefas:

$$D(S) = d_1(S) \oplus d_2(S) \oplus \cdots \oplus d_k(S) = \sum_{i=1}^{k} d_i(S)$$

Métricas de Desempenho

Exemplo em C

Incrementa um vetor de inteiros de forma concorrente

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "timer.h"

int *vet; //vetor que será manipulado no programa

typedef struct {
   short int nthreads;
   long int dim;
   short int id;
} t_args;

//função executada pelas threads
void * tarefa (void * args) {
   t_args *arg = (t_args*) args;
   long int fatia, ini, fim;

   fatia = arg->dim/arg->nthreads;
   ini = arg->id * fatia;
   fim = ini+fatia;
   if(arg->id == (arg->nthreads)-1) fim = arg->dim;

   for(long int i=ini; i<fim; i++)
      *(vet+i)+=1;

   free(args);
   pthread_exit(NULL);
}

//aloca espaço e inicializa o vetor de inteiros
short int inicia_vetor(long int dim, int **vet) {
//retorna 0 em sucesso
   (*vet) = (int*) malloc(sizeof(int)*dim);
   if((*vet)==NULL) return 1;
   for(long int i=0; i<dim; i++)
       *(*vet+i) = (int) i;
   return 0;
}

//verifica a corretude do vetor de saida
long int verifica_vetor(long int dim, int *vet) {
//retorna 0 em sucesso e  caso contrario primeira posicao que encontrou erro
   for(long int i=0; i<dim; i++)
      if(*(vet+i) != (int) i + 1) return i;
   return 0;
}


//função principal
int main(int argc, char *argv[]) {
   short int nthreads;  //qtde de threads
   long int dim; //dimensao do vetor
```

```
   double start, end, delta;

   //recebe e valida os parametros de entrada
   if(argc < 3) {
      fprintf(stderr, "ERRO de entrada, digite: <%s> <dimensao vetor> <qtde de
threads>\n", argv[0]);
      return 1;
   }
   dim = atoi(argv[1]);
   nthreads = atoi(argv[2]);
   if(nthreads>dim) nthreads=dim;

   //aloca espaço e inicializa o vetor de entrada
   if(inicia_vetor(dim, &vet)) {
      fprintf(stderr, "ERRO de inicializacao do vetor\n");
      return 1;
   }
   //aloca espaco para o vetor de tid
   pthread_t tid[nthreads];

   GET_TIME(start);//começa a contagem de tempo
   //dispara as threads
   for(short int i=0; i<nthreads; i++) {
      t_args *args = (t_args*) malloc(sizeof(t_args));
      if(args==NULL){
         fprintf(stderr, "ERRO de alocacao de argumentos da thread %hd.\n", i);
         return 2;
      }
      args->nthreads=nthreads;
      args->dim = dim;
      args->id=i;
      if(pthread_create(&tid[i], NULL, tarefa, (void*) args)) {
         fprintf(stderr, "ERRO de criacao da thread %hd.\n", i);
         return 3;
      }
   }

   //aguarda o termino das threads
   for(short int i=0; i<nthreads; i++) {
      if(pthread_join(tid[i], NULL)){
         fprintf(stderr, "ERRO de join da thread %hd.\n", i);
         return 4;
      }
   }
   GET_TIME(end); //termina a contagem de tempo
   delta = end-start; //calcula o intervalo de tempo para processamento do vetor

   //verifica a corretude da solucao concorrente
   if(verifica_vetor(dim, vet))
        fprintf(stderr, "ERRO no vetor de saida\n");
   else puts(":-)");

   printf("Tempo: %lf\n", delta);
```

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

Retorno de dados pelas threads

Capítulo 3 - Comunicação entre threads por memória compartilhada

Capítulo 4 - Sincronização por condição