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);
    }
  }
```

```c
    //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 é um problema com paralelismo de dados se D é composto de elementos de dados e é possível aplicar uma função 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 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

Na função a qual definimos a tarefa a ser executadas pelas threads, ao encerrar a thread no fim da tarefas com `pthread_exit()`, podemos passar um parâmetro que será o valor a se retornado. Com isso na função `main()` precisamos receber este valor. Quando queremos retornar mais de um valor da thread, devemos utilizar uma estrutura `struct`

```
#include <pthread.h>

//cria a estrutura de dados de retorno da thread
typedef struct {
   int idThread;
   int aux;
} t_Ret;

//funcao executada pelas threads
void* task(void* arg){
    t_Ret *ret; //estrutura de retorno

    //aloca memoria para a estrutura de retorno
    ret = malloc(sizeof(t_Ret));

    /*EXECUÇÃO DE TAREFA*/

    // Armazena os resultados
    ret->idThread = args->idThread;
    ret->aux = args->idThread * 2;

    free(arg); //libera a memoria que foi alocada na main
    pthread_exit((void*) ret);
}

int main(int argc, char* argv[]){
    // criar a alocar as variáveis

    t_Ret  *retorno; //receberá o retorno das threads (nao eh necessario alocar,
virá alocada)

    //criar as threads

    //espera todas as threads terminarem e recebe os valores retornados
   for (int i=0; i<nthreads; i++) {
     if (pthread_join(tid_sistema[i], (void**) &retorno)) {
         printf("--ERRO: pthread_join() da thread %d\n", i);
     }
     printf("Thread %d retornou %d \n", retorno->idThread, retorno->aux);
```

```
      free(retorno); //libera memoria alocada na thread
    }

     return 0;
}
```

Se quisermos, por exemplo, acumular os retornos das threads, ou seja, somar todos os valores de um vetor, basta mexer na estrutura do `for`

```
//espera todas as threads terminarem e calcula a soma total das threads
soma_par_global=0;
for(int i=0; i<nthreads; i++) {
   if (pthread_join(tid_sistema[i], (void *) &soma_retorno_threads)) {
      printf("--ERRO: pthread_join()\n"); exit(-1);
   }
   soma_par_global += *soma_retorno_threads;
   free(soma_retorno_threads);
}
```

Capítulo 3 - Comunicação entre threads por memória compartilhada

Quando uma thread precisa trocar/compartilhar uma informação com outra thread, ela o faz alterando o valor de uma variável comum para as duas threads, ou seja, uma variável que está armazenada em um endereço de memória que as duas threads compartilham (conhecem e podem acessar).

Seção críticas e ações atômicas

Normalmente esperamos que a execução das expressões aritméticas ou sentenças de atribuição da linguagem de programação sejam feitas de forma **atômica**, i.e., que uma vez iniciada a ação por uma thread ela seja executada por completo antes que outra thread opere sobre a mesma variável. Entretanto, durante a execução das sentenças ou expressões da linguagem de alto nível por threads distintas, pode ocorrer entrelaçamentos das instruções de máquina correspondentes, dando possibilidade de ocorrência de resultados inesperados.

Esse tipo de erro, gerado pela execução concorrente dos fluxos de execução que acessam uma mesma área de memória e ao menos uma das operações é de escrita, é chamado **corrida de dados**.

Toda referência a uma variável que pode ser acessada/modificada por outra thread é uma **referência crítica**. Para impedir a ocorrência de resultados indesejáveis para uma determinada computação, os trechos de código que contêm referências críticas **devem ser executados de forma atômica**. Esses trechos de código são denominados de **seção crítica do código**.

Seções de entrada e saída da seção crítica

Mecanismos de sincronização de códigos permitem **transformar uma seção crítica em uma ação atômica** usando dois outros trechos de código especiais chamados **seção de entrada e seção de saída**.

Essas duas seções adicionais de código tem por finalidade “cercar” a entrada e saída da seção crítica, garantindo que os requisitos de execução atômica sejam atendidos. O pseudocódigo abaixo ilustra essa estratégia:

```
while(true) {
executa fora da seção crítica (...)
requisita a entrada na seção crítica //seção de entrada
executa a seção crítica (...) //seção crítica
sai da seção crítica //seção de saída
}
```

Sincronização por exclusão mútua

Visa garantir que **os trechos de código em cada thread que acessam objetos compartilhados não sejam executados ao mesmo tempo**, ou que uma vez iniciados sejam executados até o fim sem que outra thread inicie a execução do trecho equivalente. Essa restrição é necessária para lidar com a possibilidade de inconsistência dos valores das variáveis compartilhadas.

A solução para a **exclusão mútua** é definida agrupando *sequências contínuas de ações atômicas de hardware* em seções críticas de software. As **seções críticas** (trechos de código que acessam objetos compartilhados) devem ser transformadas em ações atômicas, de forma que a sua execução não possa ocorrer concorrentemente com outra seção crítica que referencia a mesma variável.

- **Seção Crítica**: É o trecho de código que acessa o recurso compartilhado e que, para evitar erros, deve ser executado de forma atômica.
- **Solução: Locks (Mutex)**: A exclusão mútua é garantida usando locks (ou mutexes, do tipo `pthread_mutex_t` em Pthreads). A thread "tranca" o lock antes de entrar na seção crítica com `L.lock()` e o "destranca" ao sair com `L.unlock()`. Se uma thread tenta trancar um lock que já está em uso, ela é bloqueada até que o lock seja liberado.

sincronização por espera ocupada

A sincronização por espera ocupada faz com que a thread fique continuamente testando o valor de uma determinada variável até que esse valor lhe permita executar a sua seção crítica com exclusividade. Para implementar esse mecanismo de sincronização é necessário dispor de instruções de máquina que permitam ler e escrever em localizações da memória de forma atômica.
O principal problema da solução por espera ocupada é que ela gasta ciclos de CPU enquanto espera autorização para seguir com o seu fluxo de execução normal. A “espera ocupada” só faz sentido nos seguintes casos:
• não há nada melhor para a CPU fazer enquanto espera;
• o tempo de espera é menor que o tempo requerido para a troca de contexto entre threads.

sincronização por escalonamento (`lock`)

Sendo a alternativa mais usual, um `lock` **possui uma thread proprietária** e esta relação de posse determina características particulares das operações sobre locks:
Uma thread requisita a posse de um lock L executando a operação `L.lock()`;
Uma thread que executa a operação `L.lock()` torna-se a proprietária do lock se nenhuma outra thread já possui o lock, caso contrário a thread é bloqueada;
Um thread libera sua posse sobre o lock executando a operação `L.unlock` (se a thread não possui o lock a

operação retorna com erro);
Uma thread que já possua o lock L e executa `L.lock()` novamente não é bloqueada (apenas se o lock for recursivo), mas deve executar `L.unlock()` o mesmo número de vezes que executou `L.lock()` antes que outra thread possa ganhar a posse de L;

O uso de locks para implementar exclusão mútua se dá da seguinte forma: a operação `L.lock()` implementa a entrada na seção crítica e a operação `L.unlock()` implementa a saída da seção crítica,

Uso em C

A biblioteca Pthreads oferece o mecanismo de sincronização por locks através de variáveis especiais do tipo `pthread_mutex_t`. Por definição, Pthreads implementa locks não-recursivos (uma thread não deve tentar alocar novamente um lock que já possui). Para tornar o lock recursivo é preciso mudar suas propriedades básicas.

`pthread_mutex_t`

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// a função inicializa o mutex referenciado por mutex com atributos especificados
por attr
int pthread_mutex_init(pthread_mutex_t *mutex,const pthread_mutexattr_t *attr);

// a função destrói o objeto mutex referenciado por _mutex_
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

`lock`

```
// O objeto mutex referenciado por mutex é bloqueado
int pthread_mutex_lock(pthread_mutex_t *mutex);

// é idêntico a pthread_mutex_lock() exceto que se o objeto mutex referenciado por
mutex está atualmente bloqueado (por qualquer thread, incluindo o thread atual), a
chamada retorna imediatamente
int pthread_mutex_trylock(pthread_mutex_t *mutex);

// O objeto mutex referenciado por mutex é desbloqueado
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Código completo

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

long int soma = 0; //variavel compartilhada entre as threads
pthread_mutex_t mutex; //variavel de lock para exclusao mutua
```

```c
//funcao executada pelas threads
void *Tarefa (void *arg) {
  long int id = (long int) arg;
  printf("Thread : %ld esta executando...\n", id);

  for (int i=0; i<100000; i++) {
     //--entrada na SC
     pthread_mutex_lock(&mutex);
     //--SC (seção critica)
     soma++; //incrementa a variavel compartilhada
     //--saida da SC
     pthread_mutex_unlock(&mutex);
  }
  printf("Thread : %ld terminou!\n", id);
  pthread_exit(NULL);
}

//fluxo principal
int main(int argc, char *argv[]) {
   pthread_t *tid; //identificadores das threads no sistema
   int nthreads; //qtde de threads (passada linha de comando)

   //--le e avalia os parametros de entrada
   if(argc<2) {
      printf("Digite: %s <numero de threads>\n", argv[0]);
      return 1;
   }
   nthreads = atoi(argv[1]);

   //--aloca as estruturas
   tid = (pthread_t*) malloc(sizeof(pthread_t)*nthreads);
   if(tid==NULL) {puts("ERRO--malloc"); return 2;}

   //--inicilaiza o mutex (lock de exclusao mutua)
   pthread_mutex_init(&mutex, NULL);

   //--cria as threads
   for(long int t=0; t<nthreads; t++) {
     if (pthread_create(&tid[t], NULL, ExecutaTarefa, (void *)t)) {
       printf("--ERRO: pthread_create()\n"); exit(-1);
     }
   }

   //--espera todas as threads terminarem
   for (int t=0; t<nthreads; t++) {
     if (pthread_join(tid[t], NULL)) {
         printf("--ERRO: pthread_join() \n"); exit(-1);
     }
   }

   //--finaliza o mutex
   pthread_mutex_destroy(&mutex);
```

```
    printf("Valor de 'soma' = %ld\n", soma);

    return 0;
}
```

Capítulo 4 - Sincronização por condição

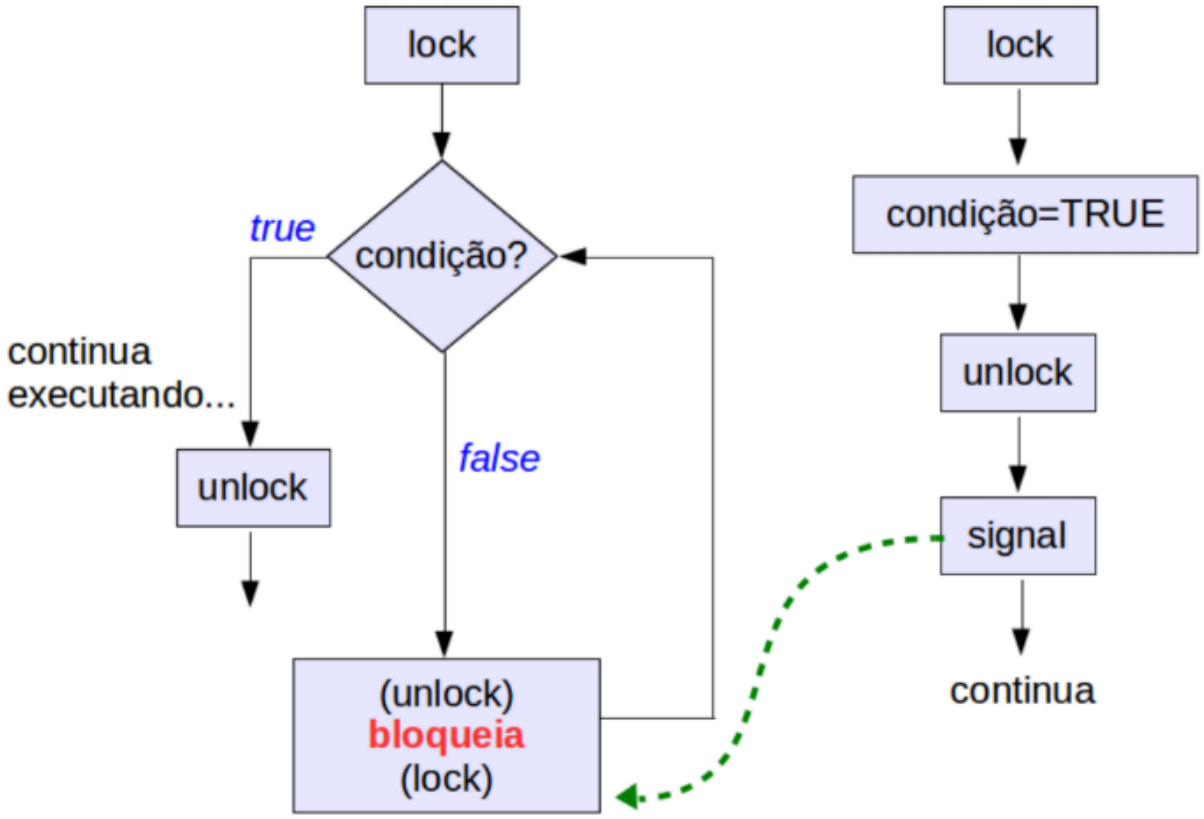
Visa garantir que **uma thread seja retardada enquanto uma determinada condição lógica da aplicação não for satisfeita**. A solução para a sincronização por condição é implementada **bloqueando a execução de uma thread até que o estado da aplicação seja correto para a sua execução**. Em resumo, para coordenar a execução de um algoritmo concorrente é necessário que os fluxos de execução troquem informações entre si sobre seus estados de execução e possam suspender ou retomar suas execuções dependendo do estado de execução dos demais fluxos ou do estado global da aplicação.

Variáveis de Condição

Permitem que as threads esperem (bloqueando-se) até que sejam sinalizadas (avisadas) por outra thread que a condição lógica foi atendida. Associado a essas variáveis de condição, temos normalmente três operações principais:

- **WAIT(condvar)**: bloqueia a thread na fila da variável de condição;
- SIGNAL(condvar): desbloqueia uma thread na fila da variável de condição;
- BROADCAST(condvar): desbloqueia todas as threads na fila da variável de condição.

Uma **variável de condição** é sempre usada em conjunto com uma variável de lock. A thread usa o bloco de lock para checar a condição lógica da aplicação e decidir por WAIT ou SIGNAL O lock é implicitamente liberado quando a thread é bloqueada e é implicitamente devolvido quando a thread retoma a execução do ponto de bloqueio.



Uso em C

A biblioteca Pthreads define um tipo especial chamado `pthread_cond_t` com as seguintes rotinas:

`pthread_cond_wait (condvar, mutex)` : bloqueia a thread na condição (condvar) (deve ser chamada com mutex locado para a thread e depois de finalizado deve desalocar mutex);

`pthread_cond_signal (condvar)` : desbloqueia uma thread esperando pela condição (condvar);

`pthread_cond_broadcast (condvar)` : usado no lugar de SIGNAL quando todas as threads na fila da condição podem ser desbloqueadas;

`pthread_cond_init (condvar, attr)` : inicializa a variável;

`pthread_cond_destroy (condvar)` : libera a variável.

Sincronização por Barreira

Um tipo particular de sincronização por condição aparecem problemas que requerem que todos os fluxos de execução (ou um grupo deles) sincronizem suas ações em determinados pontos da execução do algoritmo. Essa sincronização é chamada de **sincronização por barreira** porque funciona exatamente como uma “barreira” (ou bloqueio coletivo) que faz com que todos os fluxos de execução aguardem pelos demais até que todos tenham chegado a esse ponto (ou passo) do algoritmo.

Uso em C

```
/* Variaveis globais inicializadas na main */
pthread_mutex_t mutex;
pthread_cond_t cond;
```

```c
//funcao barreira
void barreira(int nthreads) {
    static int bloqueadas = 0;
    pthread_mutex_lock(&mutex); //inicio secao critica

    if (bloqueadas == (nthreads-1)) {
        //ultima thread a chegar na barreira
        bloqueadas=0;
        pthread_cond_broadcast(&cond);
    } else {
        bloqueadas++;
        pthread_cond_wait(&cond, &mutex);
    }
    pthread_mutex_unlock(&mutex); //fim secao critica
}

void *task (void *t) {
  int id = *(int*)t, i;
  int boba1, boba2;

  for (i=0; i < 4; i++) {
    printf("Thread %d: passo=%d\n", my_id, i);

    //sincronizacao condicional
    barreira(NTHREADS);

    /* simula uma computacao qualquer para consumir tempo... */
    boba1=100; boba2=-100; while (boba2 < boba1) boba2++;
  }
  pthread_exit(NULL);
}

int main(){

  /* Inicilaiza o mutex (lock de exclusao mutua) e a variavel de condicao */
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init (&cond, NULL);

    /*fluxo principal*/

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);

    return 0;
}
```

Resumo

- **Cenário**: É um tipo de sincronização por condição onde todas as threads de um grupo devem esperar em um ponto específico do algoritmo até que a última thread chegue.
- **Implementação**: A última thread a chegar na barreira usa `pthread_cond_broadcast` para desbloquear todas as outras threads que estavam esperando em um `pthread_cond_wait`.

- **Uso**: Comum em métodos iterativos, como o de Jacobi, onde cada passo do cálculo precisa ser concluído por todas as threads antes que a próxima iteração possa começar

Capítulo 5 - Semana 6 - Padrões Produtror/Consumidor e Leitores/Escritores

Padrão Produtor/Consumidor

- Os produtores produzem/geram novos elementos
- Os consumidores consomem/processam esses elementos
- O canal de comunicação (área de dados) oferece operações de inserção e remoção de elementos
  Produtores depositam/inserem os elementos gerados na área de dados
  Consumidores retiram/removem os elementos que devem ser processados da área de dados

Condições do problema produtor/consumidor

1. Os produtores não podem inserir novos elementos quando a área de dados já está cheia
2. Os consumidores não podem retirar elementos quando a área de dados já está vazia
3. Os elementos devem ser retirados na mesma ordem em que foram inseridos
4. Os elementos inseridos não podem ser perdidos (sobreescritos por novos elementos)
5. Um elemento só pode ser retirado uma única vez

Implementação

Produtor

```
void * produtor(void * arg) {
    tElemento elemento;
    while(REPETE) {
        elemento = produzElemento();
        Insere(elemento); //pode bloquear!
    }
    pthread_exit(NULL);
}
```

Consumidor

```
void * consumidor(void * arg) {
    tElemento elemento;
    while(REPETE) {
        elemento = Retira(); //pode bloquear!
        processaElemento(elemento);
    }
```

```
    pthread_exit(NULL);
}
```

Operações

```
void Insere (tElemento elem) {
    pthread_mutex_lock(&mutex);
    while(contador == N) {
        pthread_cond_wait(&cond_prod, &mutex);
    }
    contador++;
    Canal[posInsere] = elem;
    posInsere = (posInsere + 1) % N;
    pthread_mutex_unlock(&mutex);
    pthread_cond_signal(&cond_cons);
}


//------------------------------------------------------------------------//


tElemento Retira (void) {
    tElemento elem;
    pthread_mutex_lock(&mutex);
    while(contador == 0) {
        pthread_cond_wait(&cond_cons, &mutex);
    }
    contador--;
    elem = Canal[posRemove];
    posRemove = (posRemove + 1) % N;
    pthread_mutex_unlock(&mutex);
    pthread_cond_signal(&cond_prod);
    return elem;
}
```

Variáveis Globais

```
#define N 5 //tamanho do canal
//variaveis do problema
int Canal[N];
int contador=0, posInsere=0, posRemove=0;
//variaveis para sincronizacao
pthread_mutex_t mutex;
pthread_cond_t cond_cons, cond_prod;
```

Resumo

- **Cenário**: Threads "produtoras" geram itens e os inserem em um buffer (canal) compartilhado, enquanto threads "consumidoras" os retiram para processamento.
- **Condições de Sincronização**: Um produtor não pode inserir em um buffer cheio e um consumidor não pode retirar de um buffer vazio.
- **Implementação**: Usa-se um mutex para proteger o acesso ao buffer e duas variáveis de condição: `cond_prod` para sinalizar que o buffer não está mais cheio (acordando um produtor) e `cond_cons` para sinalizar que o buffer não está mais vazio (acordando um consumidor).

Leitores / Escritores

Uma área de dados (ex., arquivo, bloco da memória, tabela de uma banco de dados) é compartilhada entre diferentes threads que executam duas operações:
leitura: lêem o conteúdo da área de dados
escrita: escrevem conteúdo na área de dados

Condições

1. os leitores podem ler simultaneamente uma região de dados compartilhada
2. apenas um escritor pode escrever a cada instante em uma região de dados compartilhada
3. se um escritor está escrevendo, nenhum leitor pode ler a mesma região de dados compartilhada

Implementação

```
void *leitor (void *arg) {
    while(1) {
        EntraLeitura();
        //le algo...
        SaiLeitura();
        //faz outra coisa...
    }
}
void *escritor (void *arg) {
    while(1) {
        EntraEscrita();
        //escreve algo...
        SaiEscrita();
        //faz outra coisa...
    }
}

int leit=0, escr=0; //globais
void EntraLeitura() {
    pthread_mutex_lock(&mutex);
    while(escr > 0) {
        pthread_cond_wait(&cond_leit, &mutex);
    }
    leit++;
    pthread_mutex_unlock(&mutex);
}
```

```
void SaiLeitura() {
    pthread_mutex_lock(&mutex);
    leit--;
    if(leit==0) pthread_cond_signal(&cond_escr);
    pthread_mutex_unlock(&mutex);
}


int leit=0, escr=0; //globais
void EntraEscrita () {
    pthread_mutex_lock(&mutex);
    while((leit>0) || (escr>0)) {
        pthread_cond_wait(&cond_escr, &mutex);
    }
    escr++;
    pthread_mutex_unlock(&mutex);
}
void SaiEscrita () {
    pthread_mutex_lock(&mutex);
    escr--;
    pthread_cond_signal(&cond_escr);
    pthread_cond_broadcast(&cond_leit);
    pthread_mutex_unlock(&mutex);
}

int leitores=0; + variaveis sincroniza ç~ao
void AntesLeitura () {
    lock(&mutex); leitores++; unlock(&mutex);
}
void DepoisLeitura () {
    lock(&mutex); leitores--;
    if(leitores==0)
        pthread_cond_signal(&cond_escr);
    unlock(&mutex);
}
void Escreve(void * args) {
    lock(&mutex);
    while(leitores>0)
        pthread_cond_wait(&cond_escr, &mutex);
    //realiza a escrita de args (...)
    pthread_cond_signal(&cond_escr); unlock(&mutex);
}
```

Resumo

- **Cenário**: Múltiplas threads acessam uma área de dados compartilhada. Algumas apenas leem (leitoras) e outras modificam (escritoras).

- **Condições de Sincronização**: Múltiplos leitores podem acessar os dados simultaneamente. No entanto, um escritor precisa de acesso exclusivo; quando ele está escrevendo, nenhum outro escritor ou leitor pode acessar os dados.