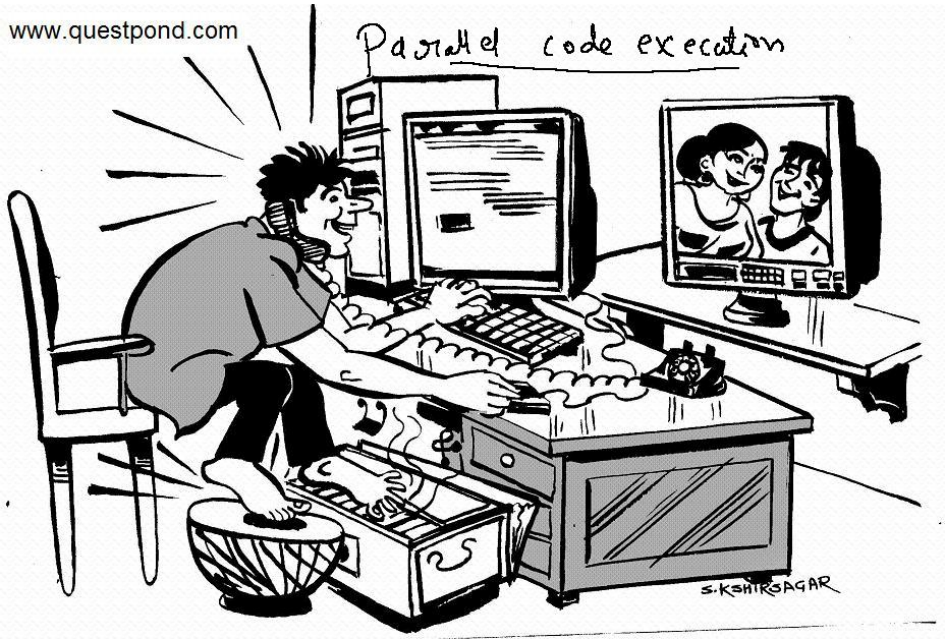
INTRODUÇÃO À PTHREADS

Isabella Vieira
Nícollas Silva
Ramon Vieira

Algoritmo e Estrutura de Dados III – 2015.1

Computação Paralela

- Programação paralela é a divisão de uma determinada aplicação em partes, de maneira que essas partes possam ser executadas simultaneamente, por vários elementos de processamento.



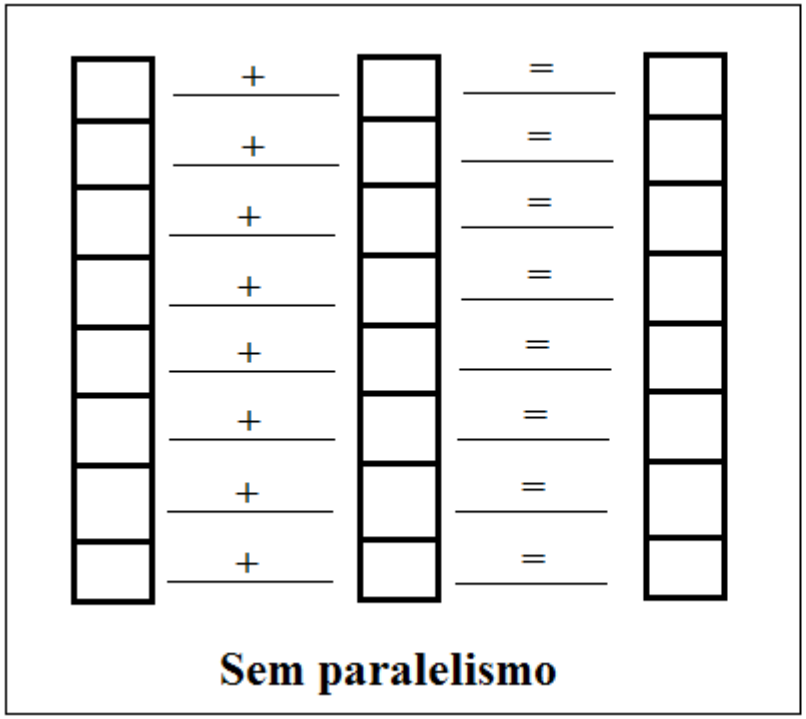
Threads

Processo = Threads + Recursos

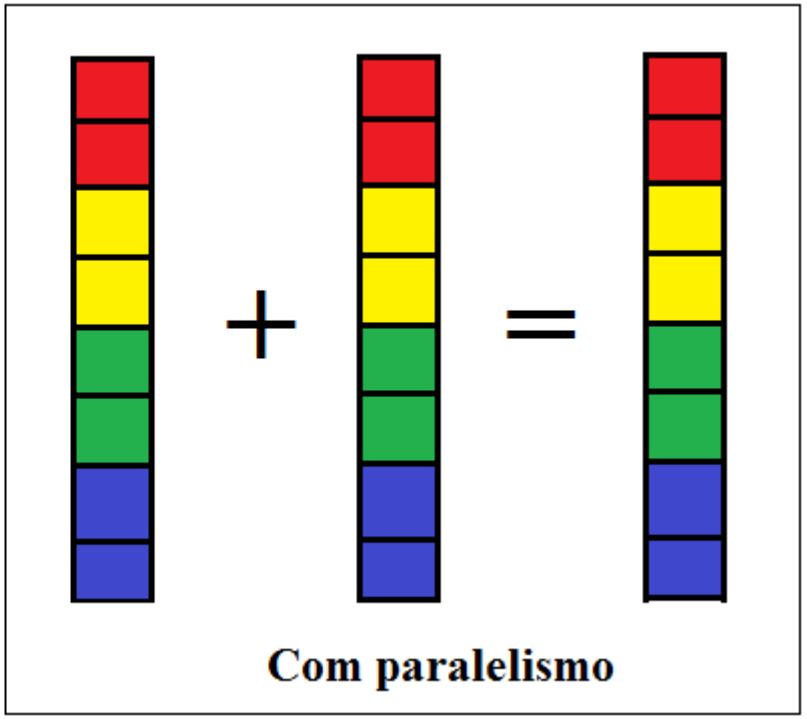
- Uma thread representa um fluxo de execução sequencial dentro do processo.
- São definidas como linhas de execução concorrentes.
- As áreas de memória do processo são compartilhados pelas threads.

Computação Paralela - Exemplo

- Deseja-se somar dois vetores distintos de oito posições.



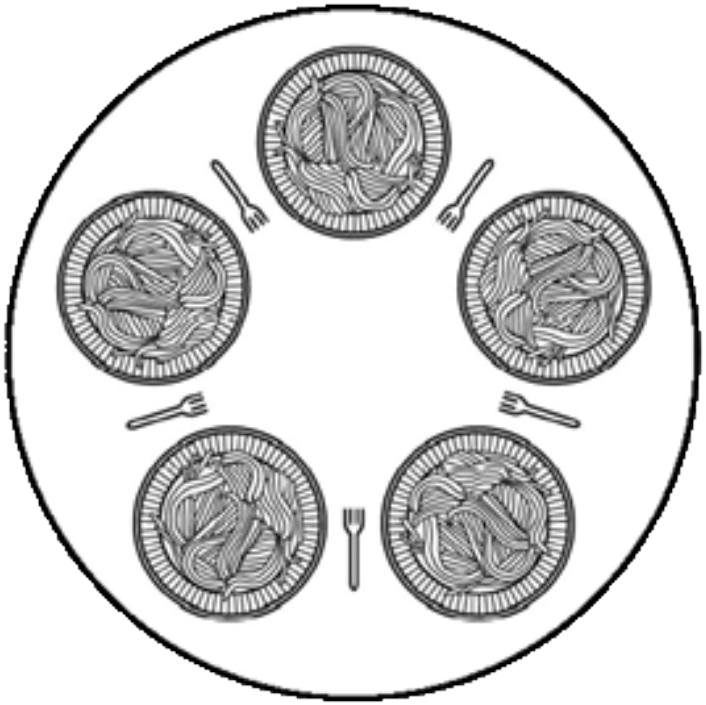
Sem paralelismo



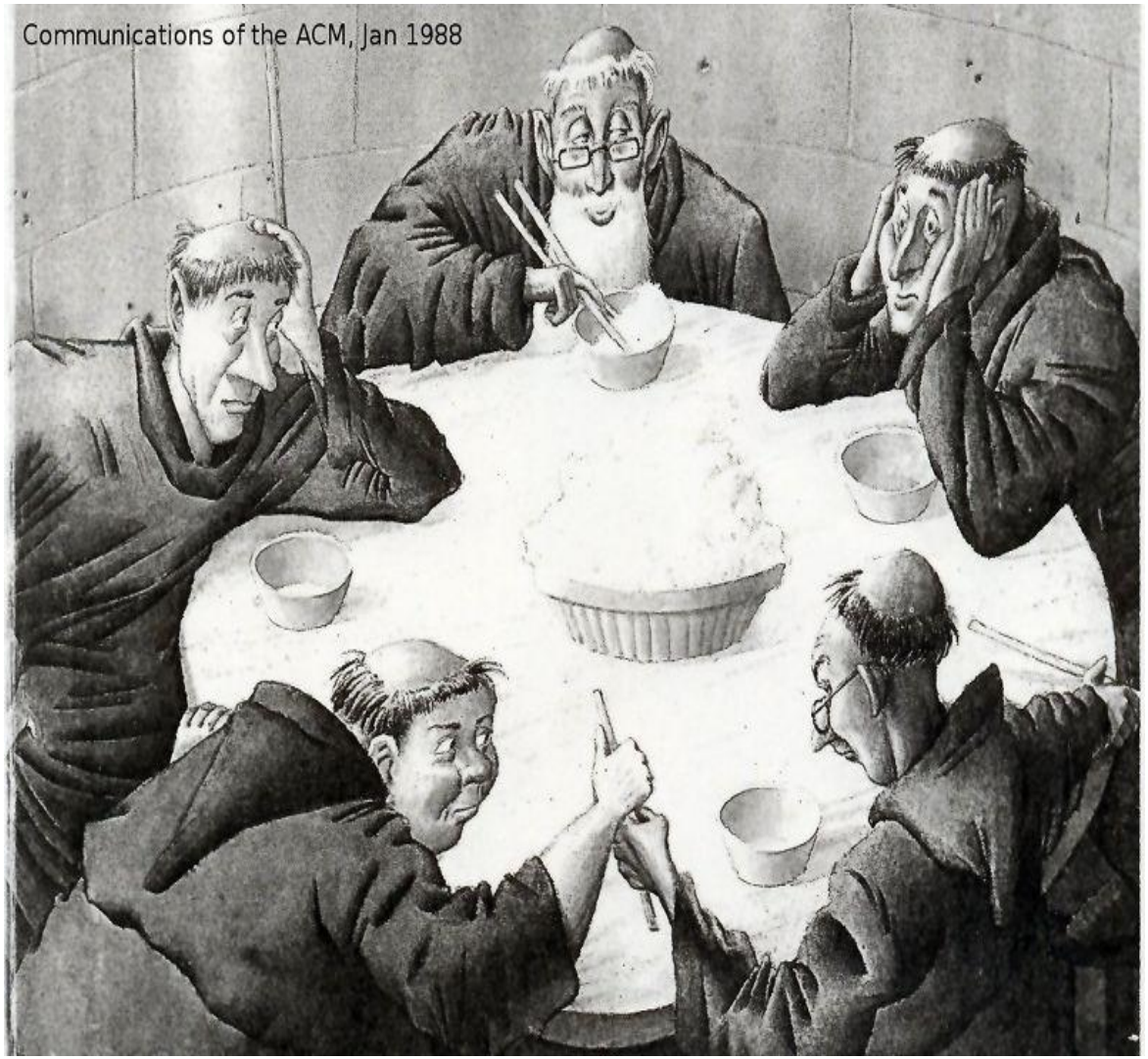
Com paralelismo

Jantar dos Filósofos - Problema

- Cinco filósofos estão sentados em uma mesa, pensando e comendo alternadamente.
- No entanto, para poderem comer o espaguete, cada filósofo precisa utilizar dois garfos e não apenas um.
- Como fazer ?? E os problemas gerados ??



Jantar dos Filósofos



Problemas com Threads

- Sincronização entre as threads
- Deadlock's
- Starvation
- Condições de corrida
- Localização de erros
- Difícil garantia de correção dos programas
- Imprevisibilidade

Computação Paralela

- Os elementos de processamento devem cooperar entre si utilizando primitivas de comunicação e sincronização, realizando a quebra do paradigma de execução sequencial do fluxo de instruções.

Trabalhando com Threads

- Como um programa pode ser desenvolvido para executar múltiplos threads dentro de um processo ?

O modelo POSIX Threads (pthreads)

- O modelo pthreads pertence à família POSIX (*Portable Operating System Interface*) e define um conjunto de rotinas para a manipulação de threads.
- As definições da biblioteca pthreads encontram-se em “pthread.h” e sua implementação em “libpthread.so”.
- Para compilar um programa com suporte a threads é necessários incluir o cabeçalho “pthread.h” e compilá-lo com “-pthread”

Criação de threads

- Quando se inicia um programa, uma thread é criada (**main thread**). Outras threads podem ser criadas usando a função:

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

- Cria uma nova thread que inicia sua execução na função indicada por start routine com o argumento indicado em arg. Em caso de sucesso, instancia th com o identificador da nova thread e retorna 0, senão retorna código de erro.

Criação de threads


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

- th é o identificador da nova thread.
- attr permite especificar como a thread deve interagir com o resto do programa.
- start routine é a função inicial que a nova thread deve executar.
- arg é o argumento a ser passado para a função start routine.

Junção de threads

- Assim como os processos, as vezes é necessário esperar que uma dada thread termine antes da execução terminar. Com processos, essa sincronização é conseguida pelas funções wait ou waitpid. Com thread a função é pthread_join.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- Suspende a execução até que a thread th termine.
- th é o identificador da thread a esperar.
- value_ptr é o valor de retorno da thread th.

Terminar threads

- Por default, existem 2 formas de uma thread terminar:
  - A função que iniciou a thread retorna.
  - A função main retorna ou alguma thread chama a função exit. Nestes dois casos todas as threads terminam.
- Outro modo de uma thread terminar é este invocar diretamente a função pthread exit

```
void pthread_exit(void *retval);
```

Exemplos práticos

- Hello world:

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

#define nthreads 2

void *func(void *arg) {
      int *argument = (int *) arg;
      printf("Olá, eu sou a thread %d\n",*argument);
}

int main() {
      int i;
      pthread_t THREADS[nthreads];
      int id[nthreads];
      while(1) {
            for (i = 0; i < nthreads; i++) {
                  id[i] = i;
                  pthread_create(&(THREADS[i]), NULL, func, &id[i]);
            }
            for (i = 0; i < nthreads; i++) {
                  pthread_join(THREADS[i], NULL);
            }
            sleep(1);
      }
      return 0;
}
```

Exemplos práticos

- Exemplo soma de vetores (serial):

```
#include <stdio.h>

#define N 10

int A[N] = {4,3,6,7,8,9,3,3,4,6};
int B[N] = {8,4,2,5,7,3,1,9,5,4};
int C[N] = {0,0,0,0,0,0,0,0,0,0};

int soma_vetor(int inicio, int fim) {
      int i;
      for (i = inicio; i < fim; i++) {
            C[i] = A[i] + B[i];
      }
}

int main() {
      soma_vetor(0, N);
      int i;
      for (i = 0; i < N; i++) {
            printf("C[%d] = %d\n", i, C[i]);
      }
      return 0;
}
```

Exemplos práticos

- Exemplo soma de vetores (paralelo):

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

#define N 10
#define nthreads 2

typedef struct {
      int inicio;
      int fim;
} thread_arg, *ptr_thread_arg;

int A[N] = {4,3,6,7,8,9,3,3,4,6};
int B[N] = {8,4,2,5,7,3,1,9,5,4};
int C[N] = {0,0,0,0,0,0,0,0,0,0};

void *func(void *arg){
      ptr_thread_arg argument = (ptr_thread_arg) arg;
      int i;

      for(i = argument->inicio ;
            i < argument->fim ; i++){
           C[i] = A[i] + B[i];
      }
}
```

```
int main(int argc, char *argv[]){
      int i;
      pthread_t THREADS[nthreads];
      thread_arg argument[nthreads];

      for(i = 0; i < nthreads; i++){
            argument[i].inicio = (i*N)/nthreads;
            argument[i].fim = ((i+1)*N)/nthreads;
            pthread_create(&(THREADS[i]), NULL,
                        func, &(argument[i]));
      }

      for(i = 0; i < nthreads; i++){
            pthread_join(THREADS[i],NULL);
      }

      for(i = 0; i < N; i++){
            printf("%d ",C[i]);
      }
      printf("\n");
      return 0;
}
```

Exemplos práticos

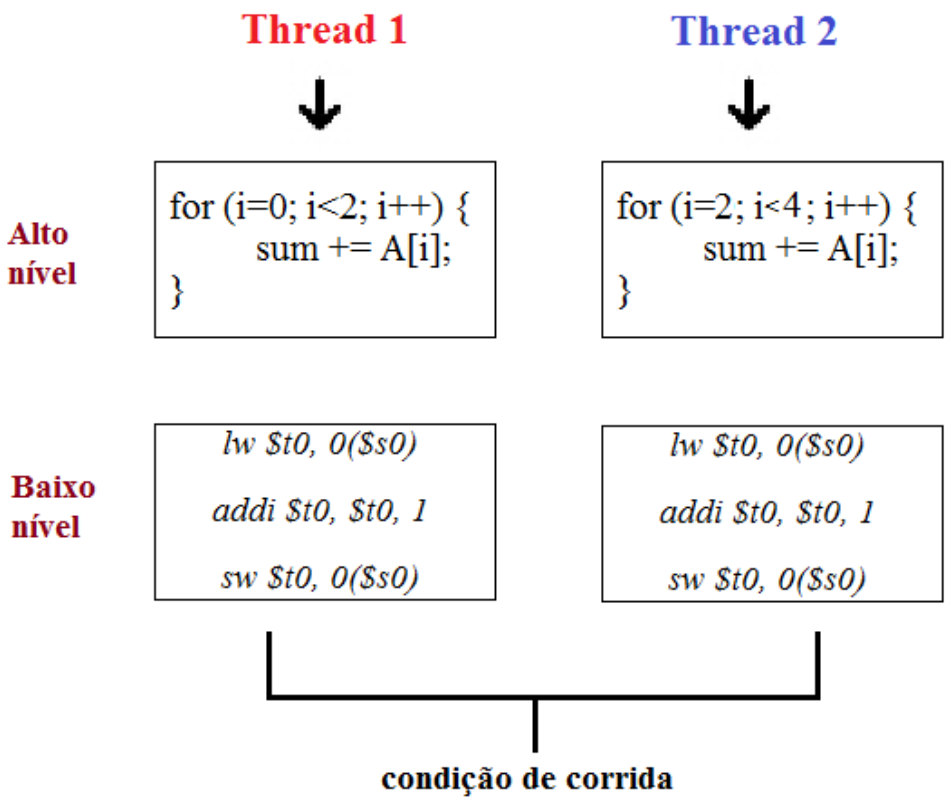
- Exemplo de soma de elementos de um vetor:

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

#define N 10
#define nthreads 2

typedef struct {
      int inicio;
      int fim;
} thread_arg, *ptr_thread_arg;

int vetor[N] = {4,3,6,7,8,9,3,3,4,6};
int soma;

void *func(void *arg){
      ptr_thread_arg argument = (ptr_thread_arg) arg;
      int i, soma_parcial;
      soma_parcial = 0;
      for(i = argument->inicio ;
             i < argument->fim ; i++){
            soma_parcial += vetor[i];
      }
      soma += soma_parcial;
}
```

```
int main(int argc, char *argv[]){
      int i;
      pthread_t THREADS[nthreads];
      thread_arg argument[nthreads];
      soma = 0;
      for(i = 0; i < nthreads; i++){
            argument[i].inicio = (i*N)/nthreads;
            argument[i].fim = ((i+1)*N)/nthreads;
            pthread_create(&(THREADS[i]), NULL,
                           func, &(argument[i]));
      }

      for(i = 0; i < nthreads; i++){
            pthread_join(THREADS[i],NULL);
      }

      printf("%d\n",soma); // resultado correto: 53

      return 0;
}
```

Problemas

- Qual o problema com a abordagem do slide anterior?
  - Operações não atômicas
  - Dados compartilhados



Problemas

- Como resolver esse problema?
  - Sincronização

- Alguns conceitos:
  - Exclusão mútua
    - uma thread está executando sozinha um determinado código, enquanto as outras esperam para poder executar
  - Sessão crítica
    - parte do programa é executada por somente uma thread de cada vez (em exclusão mútua)

Sincronização

- Primitivas de Sincronização
  - Mutexes
  - Semáforos
  - Troca de mensagens

  - Monitores (Java)

Mutex (MUTual EXclusion)

- É um lock que apenas pode estar na posse de uma thread de cada vez, garantindo exclusão mútua.

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *mutexattr);
```

- Inicia um mutex. Retorna 0 se ok, valor positivo se erro.
- mutex é variável que representa o mutex
- mutexattr permite especificar atributos do mutex. Se NULL o mutex é iniciado com atributos por padrão.
- Outra forma de iniciar um mutex (se estativamente alocado) com atributos por padrão é o seguinte

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Operações sobre Mutexes

- Obter o lock no mutex:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Liberar o lock:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Tenta obter o lock mas não bloqueia caso não seja possível:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Exemplos práticos

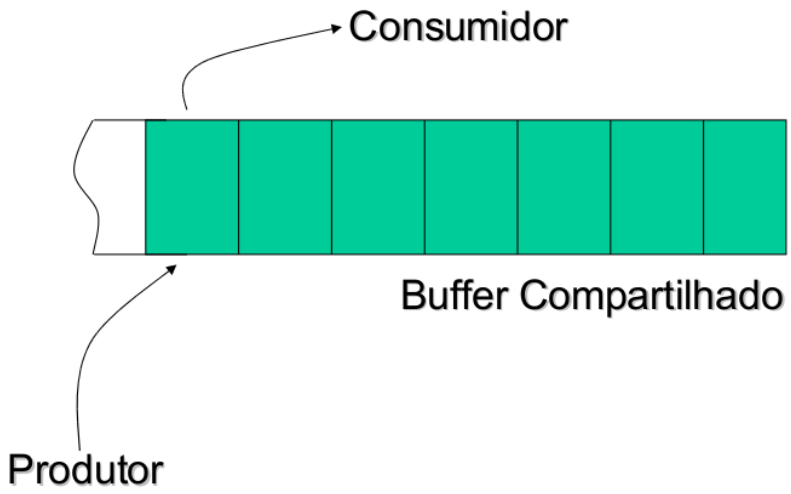
- Soma de elementos com controle de exclusão mútua:

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

#define N 10
#define nthreads 2

typedef struct {
      int inicio;
      int fim;
} thread_arg, *ptr_thread_arg;

int vetor[N] = {4,3,6,7,8,9,3,3,4,6};
int soma;
pthread_mutex_t mutex;

void *func(void *arg){
      ptr_thread_arg argument = (ptr_thread_arg) arg;
      int i, soma_parcial;
      soma_parcial = 0;
      for(i = argument->inicio ;
            i < argument->fim ; i++){
           soma_parcial += vetor[i];
      }
      pthread_mutex_lock(&mutex);
      soma += soma_parcial;
      pthread_mutex_unlock(&mutex);
}
```

```
int main(int argc, char *argv[]){
      int i;
      pthread_t THREADS[nthreads];
      thread_arg argument[nthreads];

      pthread_mutex_init(&mutex, NULL);

      for(i = 0; i < nthreads; i++){
            argument[i].inicio = (i*N)/nthreads;
            argument[i].fim = ((i+1)*N)/nthreads;
            pthread_create(&(THREADS[i]), NULL,
                           func, &(argument[i]));
      }

      for(i = 0; i < nthreads; i++){
            pthread_join(THREADS[i],NULL);
      }

      printf("%d\n",soma); // resultado correto: 53

      return 0;
}
```

Produtor-consumidor

- Dois tipos de threads que compartilham um buffer limitado:
  - Produtores: Colocam informações no buffer
  - Consumidores: Retiram informações do buffer
- Necessário sincronizar operações e controlar os acessos às variáveis.



Produtor-consumidor

- Abordagem ingênua:

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

#define NUMCONS      2
#define NUMPROD      2
#define BUFFERSIZE   1000

pthread_t cons[NUMCONS];
pthread_t prod[NUMPROD];
int buffer[BUFFERSIZE];
int prod_pos=0, cons_pos=0;

void *produtor(void *arg) {
      int n;
      while(1) {
            n = (int)(drand48() * 1000.0);
            buffer[prod_pos] = n;
            prod_pos = (prod_pos+1) % BUFFERSIZE;
            printf("Produzindo numero %d\n", n);
            sleep((int)(drand48() * 4.0));
      }
}
```

```
void *consumidor(void *arg) {
      int n;
      while(1) {
            n = buffer[cons_pos];
            cons_pos = (cons_pos+1) % BUFFERSIZE;
            printf("Consumindo numero %d\n", n);
            sleep((int)(drand48() * 4.0));
      }
}


int main(int argc, char **argv) {
  int i;
  srand48(time(NULL));
  for(i=0; i<NUMCONS; i++) {
   pthread_create(&(cons[i]), NULL, consumidor, NULL);
  }
  for(i=0; i<NUMPROD; i++) {
    pthread_create(&(prod[i]), NULL, produtor, NULL);
  }
  for(i=0; i<NUMCONS; i++) {
    pthread_join(cons[i], NULL);
  }
  for(i=0; i<NUMPROD; i++) {
    pthread_join(prod[i], NULL);
  }
}
```

Produtor-consumidor

- Possível resultado:

```
Consumindo numero 0
Consumindo numero 0
Produzindo numero 398
Produzindo numero 219
Produzindo numero 973
Produzindo numero 491
Consumindo numero 973
Consumindo numero 491
Produzindo numero 844

...
```

- Threads consumidoras consumindo itens não produzidos
- Como resolver?
  - Mutex
  - Semáforo

Produtor-consumidor

- Usando mutex:

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

#define NUMCONS      2
#define NUMPROD      2
#define BUFFERSIZE   1000

pthread_t cons[NUMCONS];
pthread_t prod[NUMPROD];
pthread_mutex_t buffer_mutex;
int buffer[BUFFERSIZE];
int prod_pos=0, cons_pos=0;

void *produtor(void *arg) {
      int n;
      while(1) {
            n = (int)(drand48() * 1000.0);
            pthread_mutex_lock(&buffer_mutex);
            buffer[prod_pos] = n;
            prod_pos = (prod_pos+1) % BUFFERSIZE;
            pthread_mutex_unlock(&buffer_mutex);
            printf("Produzindo numero %d\n", n);
            sleep((int)(drand48() * 4.0));
      }
}
```

```
void *consumidor(void *arg) {
      int n;
      while(1) {
            pthread_mutex_lock(&buffer_mutex);
            n = buffer[cons_pos];
            cons_pos = (cons_pos+1) % BUFFERSIZE;
            pthread_mutex_unlock(&buffer_mutex);
            printf("Consumindo numero %d\n", n);
            sleep((int)(drand48() * 4.0));
      }
}


int main(int argc, char **argv) {
  int i;
  srand48(time(NULL));
  pthread_mutex_init(&buffer_mutex, NULL);
  for(i=0; i<NUMCONS; i++) {
   pthread_create(&(cons[i]), NULL, consumidor, NULL);
  }
  for(i=0; i<NUMPROD; i++) {
    pthread_create(&(prod[i]), NULL, produtor, NULL);
  }
  for(i=0; i<NUMCONS; i++) {
    pthread_join(cons[i], NULL);
  }
  for(i=0; i<NUMPROD; i++) {
    pthread_join(prod[i], NULL);
  }
}
```

Produtor-consumidor

- Resolveu o problema?

```
Consumindo numero 0
Consumindo numero 0
Consumindo numero 0
Produzindo numero 463
Produzindo numero 886
Consumindo numero 0
Produzindo numero 673
Produzindo numero 370
Produzindo numero 319
Consumindo numero 319
Produzindo numero 81
Produzindo numero 25
Consumindo numero 81
Consumindo numero 25

...
```

Semáforo

- Controla o acesso à recursos compartilhados.
- Um semáforo é um inteiro não negativo com duas operações atômicas: **DOWN** (ou SLEEP ou WAIT) e **UP** (ou WAKEUP ou POST)

```
DOWN(s) {
    if (s=0)      // suspende a execução do processo
    s--;
}
```

```
UP(s) {
    s++;
    if (s=1)      // retorna um processo suspenso em s
}
```

Operações sobre Semáforos

- Cabeçalho a ser incluído:

```
#include <semaphore.h>
```

- Inicializar um semáforo:

```
int sem_init (sem_t *sem, 0, int value);
```

- DOWN:

```
int sem_wait (sem_T *sem);
```

- UP:

```
int sem_post (sem_T *sem);
```

Produtor-consumidor (semáforo)


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

#define NUMCONS      2
#define NUMPROD      2
#define BUFFERSIZE   1000

pthread_t cons[NUMCONS];
pthread_t prod[NUMPROD];
pthread_mutex_t buffer_mutex;
int buffer[BUFFERSIZE];
int prod_pos=0, cons_pos=0;
sem_t free_positions, filled_positions;

void *produtor(void *arg) {
      int n;
      while(1) {
            n = (int)(drand48() * 1000.0);
            sem_wait(&free_positions);
            pthread_mutex_lock(&buffer_mutex);
            buffer[prod_pos] = n;
            prod_pos = (prod_pos+1) % BUFFERSIZE;
            pthread_mutex_unlock(&buffer_mutex);
            sem_post(&filled_positions);
            printf("Produzindo numero %d\n", n);
            sleep((int)(drand48() * 4.0));
      }
}
```

```
void *consumidor(void *arg) {
      int n;
      while(1) {
            sem_wait(&filled_positions);
            pthread_mutex_lock(&buffer_mutex);
            n = buffer[cons_pos];
            cons_pos = (cons_pos+1) % BUFFERSIZE;
            pthread_mutex_unlock(&buffer_mutex);
            sem_post(&free_positions);
            printf("Consumindo numero %d\n", n);
            sleep((int)(drand48() * 4.0));
      }
}


int main(int argc, char **argv) {
  int i;
  srand48(time(NULL));
  pthread_mutex_init(&buffer_mutex, NULL);
  sem_init(&free_positions, 0, BUFFERSIZE);
  sem_init(&filled_positions, 0, 0);
  for(i=0; i<NUMCONS; i++) {
   pthread_create(&(cons[i]), NULL, consumidor, NULL);
  }
  for(i=0; i<NUMPROD; i++) {
    pthread_create(&(prod[i]), NULL, produtor, NULL);
  }
  for(i=0; i<NUMCONS; i++) {
    pthread_join(cons[i], NULL);
  }
  for(i=0; i<NUMPROD; i++) {
    pthread_join(prod[i], NULL);
  }
}
```

Para casa .....

- Mais informações sobre as bibliotecas:
  - http://www.google.com
  - man pthreads
- Slide complementar (jantar dos filósofos):
  - http://wiki.icmc.usp.br/images/7/76/Aula06_2.pdf

Dica para o trabalho: Profiling

- Mensura o uso de memória em funções do programa e a frequência/duração da chamada de tais funções.
- Verificar explicação no post do Isac no grupo da Computação no Facebook.

Tudo ok ?

Referências

- Slide Rafael Sachetto
  - Programação Paralela - Memória compartilhada e pthreads
- Slide Bruno Diniz
  - Programando com pthreads - DCC/UFMG
- Slide Ricardo Rocha
  - Computação Paralela - DCC/FCUP

INTRODUÇÃO À PTHREADS

Isabella Vieira
Nícollas Silva
Ramon Vieira

Algoritmo e Estrutura de Dados III – 2015.1