Sistemas Operacionais
Concorrência

Carlos Ferraz (cagf@cin.ufpe.br)
Jorge Cavalcanti Fonsêca (jcbf@cin.ufpe.br)

POSIX Threads

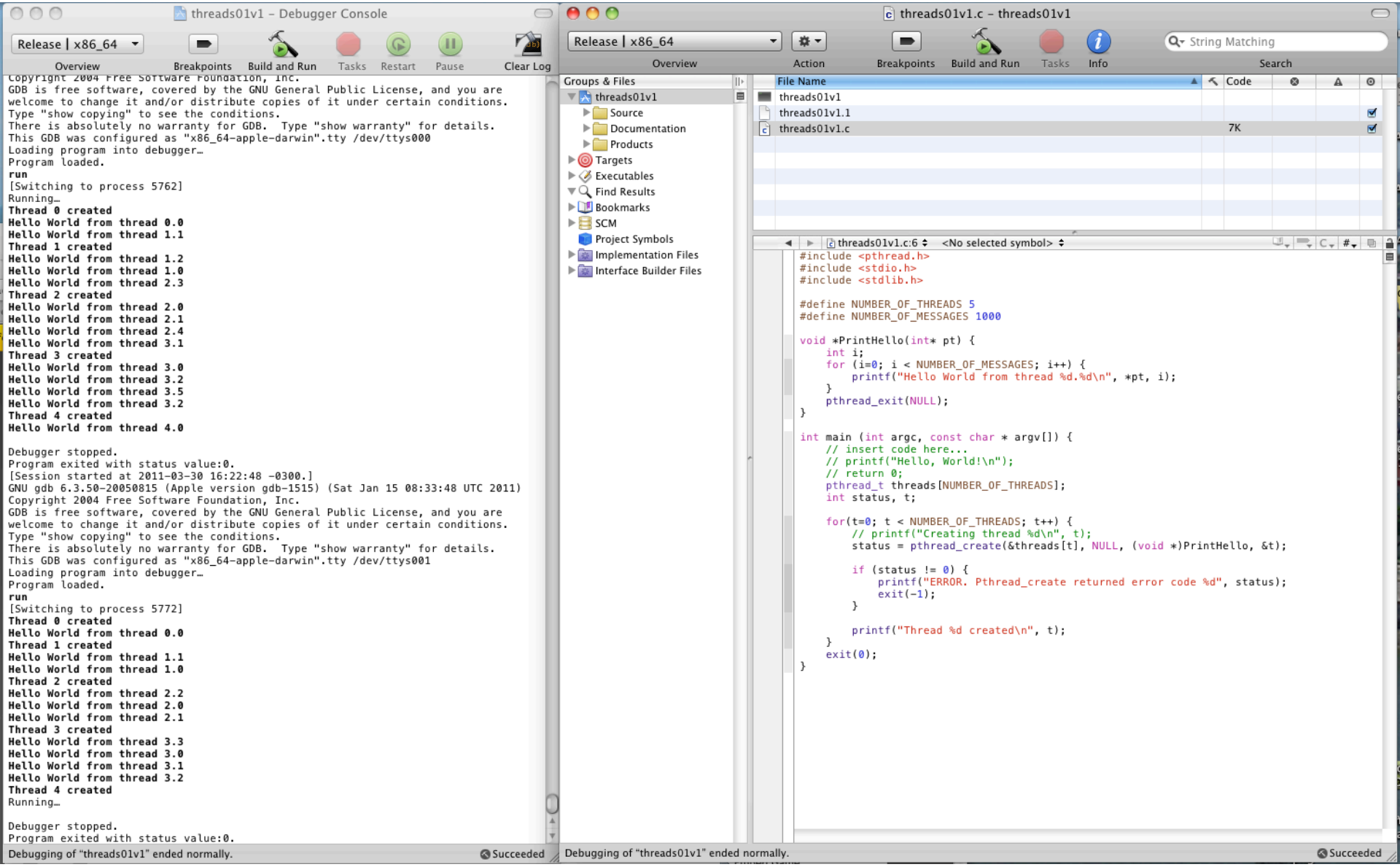
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d0, tid);
        pthread_exit(NULL);
}

        . . .
```

POSIX Threads (2)

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

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
      /* This function prints the thread's identifier and then exits. */
      printf("Hello World. Greetings from thread %d0, tid);
      pthread_exit(NULL);
}

      int main(int argc, char *argv[])
      {
            /* The main program creates 10 threads and then exits. */
            pthread_t threads[NUMBER_OF_THREADS];
            int status, i;

            for(i=0; i < NUMBER_OF_THREADS; i++) {
                  printf("Main here. Creating thread %d0, i);
                  status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                  if (status != 0) {
                        printf("Oops. pthread_create returned error code %d0, status);
                        exit(-1);
                  }
            }
            exit(NULL);
      }
```

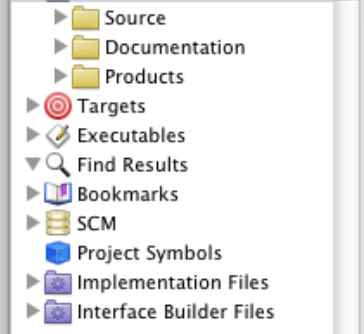
Exercício

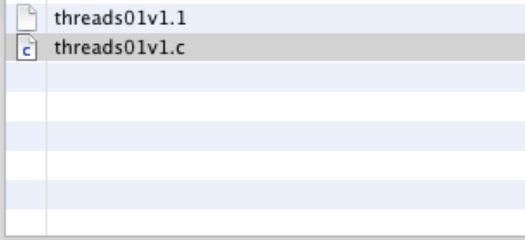
- Executando **n** vezes e verificando a ordem de execução das *threads* criadas …


threads01v1 – Debugger Console
Release | x86_64
Overview
Breakpoints
Build and Run
Tasks
Restart
Pause
Clear Log
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys000
Loading program into debugger…
Program loaded.
run
[Switching to process 5762]
Running…
Thread 0 created
Hello World from thread 0.0
Hello World from thread 1.1
Thread 1 created
Hello World from thread 1.2
Hello World from thread 1.0
Hello World from thread 2.3
Thread 2 created
Hello World from thread 2.0
Hello World from thread 2.1
Hello World from thread 2.4
Hello World from thread 3.1
Thread 3 created
Hello World from thread 3.0
Hello World from thread 3.2
Hello World from thread 3.5
Hello World from thread 3.2
Thread 4 created
Hello World from thread 4.0
Debugger stopped.
Program exited with status value:0.
[Session started at 2011-03-30 16:22:48 -0300.]
GNU gdb 6.3.50-20050815 (Apple version gdb-1515) (Sat Jan 15 08:33:48 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys001
Loading program into debugger…
Program loaded.
run
[Switching to process 5772]
Thread 0 created
Hello World from thread 0.0
Thread 1 created
Hello World from thread 1.1
Hello World from thread 1.0
Thread 2 created
Hello World from thread 2.2
Hello World from thread 2.0
Hello World from thread 2.1
Thread 3 created
Hello World from thread 3.3
Hello World from thread 3.0
Hello World from thread 3.1
Hello World from thread 3.2
Thread 4 created
Running…
Debugger stopped.
Program exited with status value:0.
Debugging of "threads01v1" ended normally.
Succeeded
threads01v1.c – threads01v1
Release | x86_64
Overview
Action
Breakpoints
Build and Run
Tasks
Info
String Matching
Search
Groups & Files
threads01v1
Source
Documentation
Products
Targets
Executables
Find Results
Bookmarks
SCM
Project Symbols
Implementation Files
Interface Builder Files
File Name
Code
threads01v1
threads01v1.1
threads01v1.c
7K
threads01v1.c:6
<No selected symbol>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMBER_OF_THREADS 5
#define NUMBER_OF_MESSAGES 1000
void *PrintHello(int* pt) {
int i;
for (i=0; i < NUMBER_OF_MESSAGES; i++) {
printf("Hello World from thread %d.%d\n", *pt, i);
}
pthread_exit(NULL);
}
int main (int argc, const char * argv[]) {
// insert code here...
// printf("Hello, World!\n");
// return 0;
pthread_t threads[NUMBER_OF_THREADS];
int status, t;
for(t=0; t < NUMBER_OF_THREADS; t++) {
// printf("Creating thread %d\n", t);
status = pthread_create(&threads[t], NULL, (void *)PrintHello, &t);
if (status != 0) {
printf("ERROR. Pthread_create returned error code %d", status);
exit(-1);
}
printf("Thread %d created\n", t);
}
exit(0);
}
Debugging of "threads01v1" ended normally.
Succeeded

```
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys000
Loading program into debugger…
Program loaded.
run
[Switching to process 5762]
Running…
Thread 0 created
Hello World from thread 0.0
Hello World from thread 1.1
Thread 1 created
Hello World from thread 1.2
Hello World from thread 1.0
Hello World from thread 2.3
Thread 2 created
Hello World from thread 2.0
Hello World from thread 2.1
Hello World from thread 2.4
Hello World from thread 3.1
Thread 3 created
Hello World from thread 3.0
Hello World from thread 3.2
Hello World from thread 3.5
Hello World from thread 3.2
Thread 4 created
Hello World from thread 4.0

Debugger stopped.
Program exited with status value:0.
[Session started at 2011-03-30 16:22:48 -0300.]
GNU gdb 6.3.50-20050815 (Apple version gdb-1515) (Sat Jan 15 08:33:48 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys001
Loading program into debugger…
Program loaded.
run
[Switching to process 5772]
Thread 0 created
Hello World from thread 0.0
Thread 1 created
Hello World from thread 1.1
Hello World from thread 1.0
Thread 2 created
Hello World from thread 2.2
Hello World from thread 2.0
Hello World from thread 2.1
Thread 3 created
Hello World from thread 3.3
Hello World from thread 3.0
Hello World from thread 3.1
Hello World from thread 3.2
Thread 4 created
Running…

Debugger stopped.
Program exited with status value:0.
```

Debugging of "threads01v1" ended normally. Succeeded



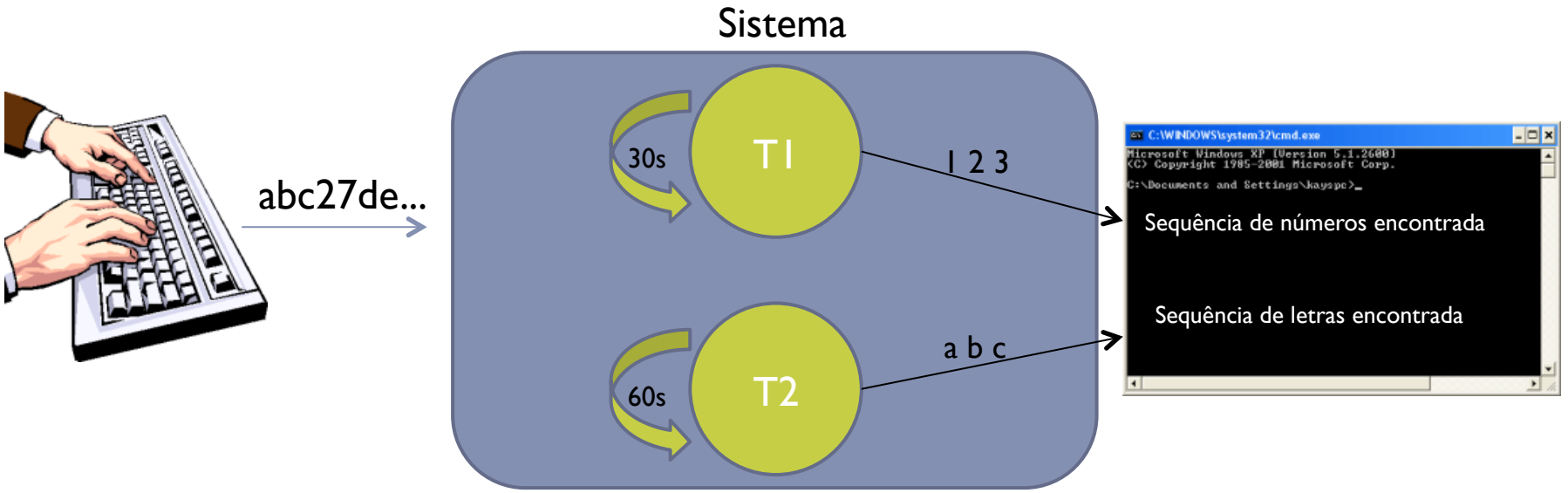


threads01v1.c:6 <No selected symbol>

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

#define NUMBER_OF_THREADS 5
#define NUMBER_OF_MESSAGES 1000

void *PrintHello(int* pt) {
    int i;
    for (i=0; i < NUMBER_OF_MESSAGES
        printf("Hello World from thr
    }
    pthread_exit(NULL);
}

int main (int argc, const char * arg
    // insert code here...
    // printf("Hello, World!\n");
    // return 0;
    pthread_t threads[NUMBER_OF_THRE
    int status, t;

    for(t=0; t < NUMBER_OF_THREADS;
        // printf("Creating thread %
        status = pthread_create(&thr

        if (status != 0) {
            printf("ERROR. Pthread_c
            exit(-1);
        }

        printf("Thread %d created\n"
    }
    exit(0);
}
```

Debugging of "threads01v1" ended normally.

Threads em Java- Exemplo



Cada thread detecta e remove a sequência encontrada (uma para números, outra para letras)

Não existe interação entre as threads

Até existe uma comunicação de informação/dados

Centro de Informática UFPE

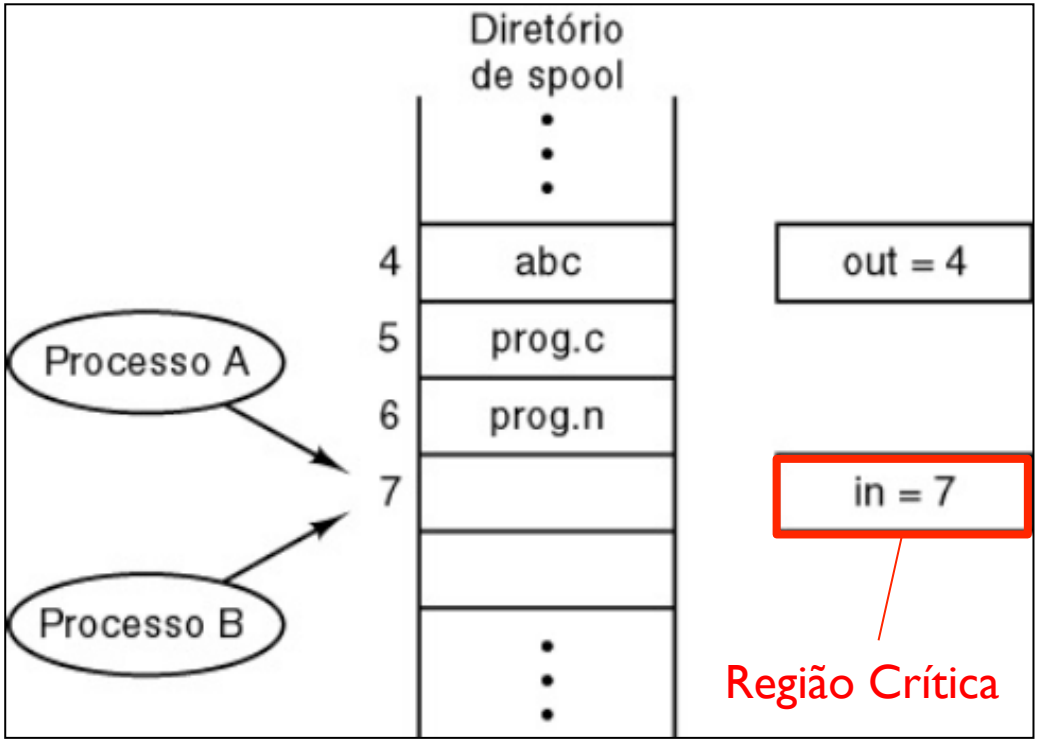
Comunicação entre processos

- Processos precisam se comunicar
  - Processo produz um valor que outro processo precisa usar
    - Ex. *pipeline do shell*

- IPC (*Interprocess Communication*)
  - 3 "Pilares"
    - Como passar informação
      - Na prática não existe no contexto de *Threads*
    - Como não entrar em conflito
      - também em *Threads*
    - Como "controlar" dependências
      - também em *Threads*

IPC

- Caso Impressora
  - *Diretório de Spool*
  - *2 processos*
    - *O processo que quer imprimir*
    - *Daemon de impressão*



2 processos querem acessar uma memória compartilhada ao mesmo tempo

***Race conditions* (Condições de Corrida)**
Resultado depende de quem executa

Precisamos de *Mutual Exclusion* (Exclusão Mútua)

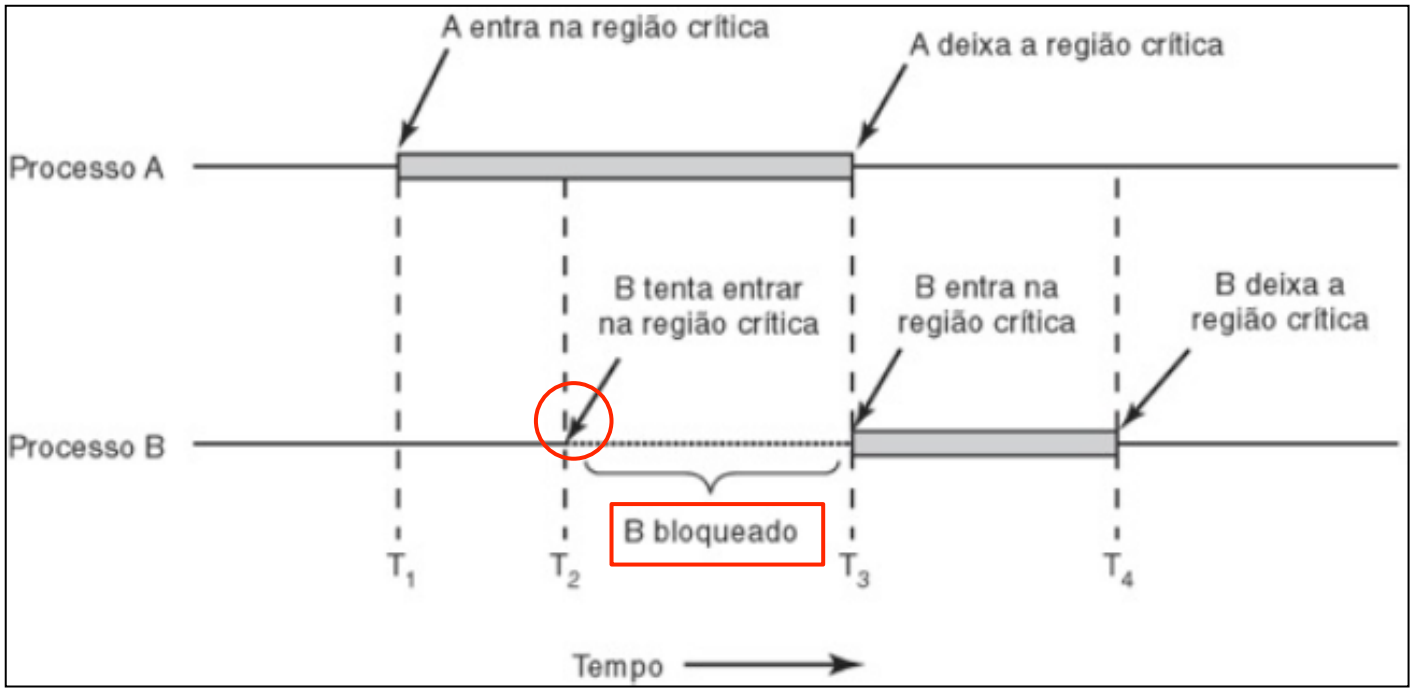
Região Crítica

- Parte do programa em que há acesso à memória compartilhada
- Se 2 processos nunca estiverem em suas regiões críticas ao mesmo tempo, é possível evitar as "disputas".

Região Crítica

- Quatro condições necessárias para prover exclusão mútua
  1. *Nunca dois processos simultaneamente em uma região crítica*
  2. *Não se pode considerar velocidades ou números de CPUs*
  3. *Nenhum processo executando fora de sua região crítica pode bloquear outros processos*
  4. *Nenhum processo deve esperar eternamente para entrar em sua região crítica*

Região Crítica



Exclusão Mútua usando regiões críticas

Exclusão Mútua

- Desabilitar interrupções
  - *Seria prudente dar aos processos de usuários o poder de desligar interrupções?*
    - *E se algum processo desabilitar e nunca mais habilitar de volta?*

      Apenas o S.O. faz isso (apenas 1 processador)
  - *E se o sistema for Multi-processador?*
    - *Desabilitar afetará somente a CPU que executou a instrução*
- Ao invés de desabilitar interrupção, é válido usar uma variável de trava (lock)?

Exclusão Mútua

- Chaveamento obrigatório

turn = 0
turn = 1
turn = 0
turn = 1

```
while (TRUE) {
    while (turn !=0);          /* laço */
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn !=1);          /* laço */
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Solução proposta para o problema da região crítica

Espera Ociosa (*Busy waiting*) – Como fica a CPU nesse momento?
Trava giratória – *Spin Lock*

Viola condição 3
(*Nenhum processo executando fora de sua região crítica pode bloquear outros processos***)**

Exclusão Mútua (Peterson)

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* número de processos */

int turn;                               /* de quem é a vez? */
int interested[N];                      /* todos os valores inicialmente em 0 (FALSE) */

void enter_region(int process);         /* processo é 0 ou 1 */
{
     int other;                         /* número de outro processo */

     other = 1 – process;               /* o oposto do processo */
     interested[process] = TRUE;        /* mostra que você está interessado */
     turn = process;                    /* altera o valor de turn */
     while (turn == process && interested[other] == TRUE) /* comando nulo */;
}

void leave_region(int process)          /* processo: quem está saindo */
{
     interested[process] = FALSE;       /* indica a saída da região crítica */
}
```

Solução de Peterson (1981) para exclusão mútua

O que acontece se 2 processos chamam *enter_region(int)* ao mesmo tempo?

Exclusão Mútua (Peterson)

Processo 0

```
void enter_region(int process);
{
    int other;

    other = 1 – process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE)
}
```

Processo 1

```
void enter_region(int process);
{
    int other;

    other = 1 – process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE)
}
```

```
void leave_region(int process)
{
        interested[process] = FALSE;
}
```

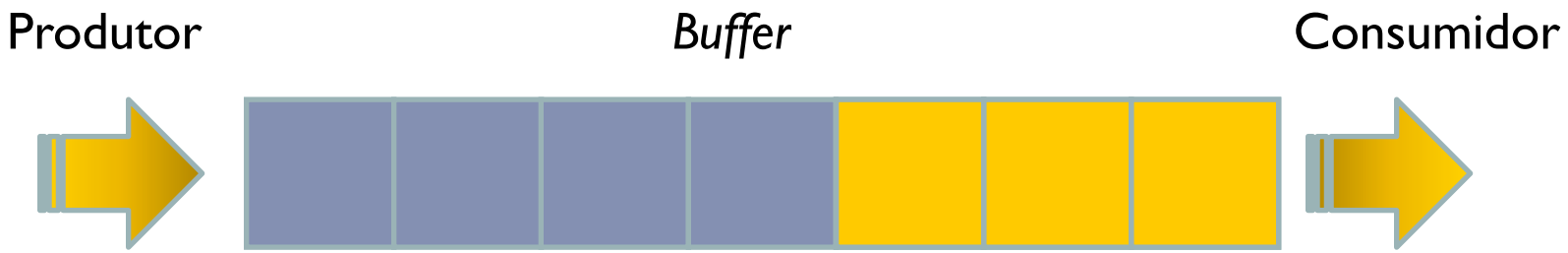
...

Também com Espera Ociosa (*Busy waiting*)

Exclusão Mútua

- Espera ociosa
  - *Quando quer entrar em sua região critica, um processo verifica se sua entrada é permitida. Se não for, ele ficará em um laço esperando (ocioso) até que seja permitida a entrada.*
  - *Efeitos inesperados*
    - *Problema da inversão de prioridade*
      - *Processos: alta e baixa prioridade*
      - *Escalonamento: alta prioridade no estado pronto: executa*
- *É preciso "dormir" e "acordar"*
  - *Primitivas do S.O.*

Problema do Produtor-Consumidor



- se consumo > produção
  - Buffer esvazia; Consumidor não tem o que consumir
- se consumo < produção
  - Buffer enche; Produtor não consegue produzir mais
- Problema clássico
  - também conhecido como Buffer Limitado
  - Uso das primitivas *sleep* e *wakeup*!

Problema do Produtor-Consumidor

```
#define N 100                                   /* número de lugares no buffer */
int count = 0;                                  /* número de itens no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* número de itens no buffer */
          item = produce_item( );               /* gera o próximo item */
          if (count == N) sleep( );             /* se o buffer estiver cheio, vá dormir */
          insert_item(item);                    /* ponha um item no buffer */
          count = count + 1;                    /* incremente o contador de itens no buffer */
          if (count == 1) wakeup(consumer);     /* o buffer estava vazio? */
     }
}
```

Problema do Produtor-Consumidor

```
void consumer(void)
{
     int item;

     while (TRUE) {                                  /* repita para sempre */
          if (count == 0) sleep( );                  /* se o buffer estiver vazio, vá dormir */
          item = remove_item( );                     /* retire o item do buffer */
          count = count – 1;                         /* decresça de um o contador de itens no buffer */
          if (count == N – 1) wakeup(producer);      /* o buffer estava cheio? */
          consume_item(item);                        /* imprima o item */
     }
}
```

Problema do Produtor-Consumidor

Produtor

```
while (TRUE) {                          /* número de itens no buffer */
    item = produce_item( );             /* gera o próximo item */
    if (count == N) sleep( );           /* se o buffer estiver cheio, vá dormir */
    insert_item(item);                  /* ponha um item no buffer */
    count = count + 1;                  /* incremente o contador de itens no buffer */
    if (count == 1) wakeup(consumer);   /* o buffer estava vazio? */
}
```

Porém... **Disputa fatal** pode acontecer

Consumidor lendo count = 0

e escalonador troca de processo (similar ao Spool) ...

Ambos dormirão para sempre !!!

bit de espera pelo sinal de acordar (*wakeup waiting bit*)

Consumidor

```
while (TRUE) {                          /* repita para sempre */
    if (count == 0) sleep( );           /* se o buffer estiver vazio, vá dormir */
    item = remove_item( );              /* retire o item do buffer */
    count = count – 1;                  /* decresça de um o contador de itens no buffer */
    if (count == N – 1) wakeup(producer);  /* o buffer estava cheio? */
    consume_item(item);                 /* imprima o item */
}
```

Semáforos

- **Semáforo** é uma variável que tem como função o controle de acesso a recursos compartilhados

- Evolução do *bit de espera pelo sinal de acordar*
  - Contador no lugar do bit

- As operações de incrementar e decrementar devem ser operações **atômicas**, ou **indivisíveis**, ou seja,
  - enquanto um processo estiver executando uma dessas duas operações, nenhum outro processo pode executar outra operação sob o mesmo semáforo, devendo esperar que o primeiro processo encerre sua operação.
  - Essa obrigação evita **condições de disputa** entre vários processos

Primitivas de S.O. → com auxílio de Hardware

Produtor-Consumidor com Semáforos

```
#define N 100                                   /* número de lugares no buffer */
typedef int semaphore;                          /* semáforos são um tipo especial de int */
:
:
:

void producer(void)
{
      int item;

      while (TRUE) {                            /* TRUE é a constante 1 */
            item = produce_item( );             /* gera algo para pôr no buffer */


            insert_item(item);                  /* põe novo item no buffer */


      }
}
```

Produtor-Consumidor com Semáforos

```
void consumer(void)
{
     int item;

     while (TRUE) {                          /* laço infinito */
          down(&full);                       /* decresce o contador full */
                                             /* entra na região crítica */
          item = remove_item( );             /* pega o item do buffer */
                                             /* deixa a região crítica */
          up(&empty);                        /* incrementa o contador de lugares vazios */
          consume_item(item);                /* faz algo com o item */
     }
}
```

Produtor-Consumidor com Semáforos

```
#define N 100                                   /* número de lugares no buffer */
typedef int semaphore;                          /* semáforos são um tipo especial de int */
                                                /* controla o acesso à região crítica */
                                                /* conta os lugares vazios no buffer */
                                                /* conta os lugares preenchidos no buffer */

void producer(void)
{
      int item;

      while (TRUE) {                            /* TRUE é a constante 1 */
            item = produce_item( );             /* gera algo para pôr no buffer */
                                                /* decresce o contador empty */
                                                /* entra na região crítica */
            insert_item(item);                  /* põe novo item no buffer */
                                                /* sai da região crítica */
                                                /* incrementa o contador de lugares preenchidos */
      }
}
```

Produtor-Consumidor com Semáforos

```
void consumer(void)
{
     int item;

     while (TRUE) {                          /* laço infinito */
                                             /* decresce o contador full */
          down(&mutex);                      /* entra na região crítica */
          item = remove_item( );             /* pega o item do buffer */
          up(&mutex);                        /* deixa a região crítica */
                                             /* incrementa o contador de lugares vazios */
          consume_item(item);                /* faz algo com o item */
     }
}
```

Mutex (Semáforo simplificado/binário)

Produtor-Consumidor com Semáforos

Produtor

```
while (TRUE) {                          /* TRUE é a constante 1 */
        item = produce_item( );         /* gera algo para pôr no buffer */
        down(&empty);                   /* decresce o contador empty */
        down(&mutex);                   /* entra na região crítica */
        insert_item(item);              /* põe novo item no buffer */
        up(&mutex);                     /* sai da região crítica */
        up(&full);                      /* incrementa o contador de lugares preenchidos */
}
```

Consumidor

```
while (TRUE) {                          /* laço infinito */
        down(&full);                    /* decresce o contador full */
        down(&mutex);                   /* entra na região crítica */
        item = remove_item( );          /* pega o item do buffer */
        up(&mutex);                     /* deixa a região crítica */
        up(&empty);                     /* incrementa o contador de lugares vazios */
        consume_item(item);             /* faz algo com o item */
}
```

Mutex (Semáforo simplificado)

Chamada de thread	Descrição
pthread_mutex_init	Cria um mutex
pthread_mutex_destroy	Destrói um mutex existente
pthread_mutex_lock	Conquista uma trava ou bloqueio
pthread_mutex_trylock	Conquista uma trava ou falha
pthread_mutex_unlock	Libera uma trava

Tabela 2.6 Algumas chamadas de Pthreads relacionadas a mutexes.

pThreads - Linux

Mutex (Semáforo simplificado)

Chamada de thread	Descrição
pthread_cond_init	Cria uma variável de condição
pthread_cond_destroy	Destrói uma variável de condição
pthread_cond_wait	Bloqueio esperando por um sinal
pthread_cond_signal	Sinaliza para outro thread e o desperta
pthread_cond_broadcast	Sinaliza para múltiplos threads e desperta todos eles

Tabela 2.7 Algumas chamadas de Pthreads relacionadas a variáveis de condição.

pThreads - Linux

Monitores (1)

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Exemplo de um monitor

Monitores (2)

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

- O problema do produtor-consumidor com monitores
  - somente um procedimento está ativo por vez no monitor
  - o buffer tem N lugares

Barreiras

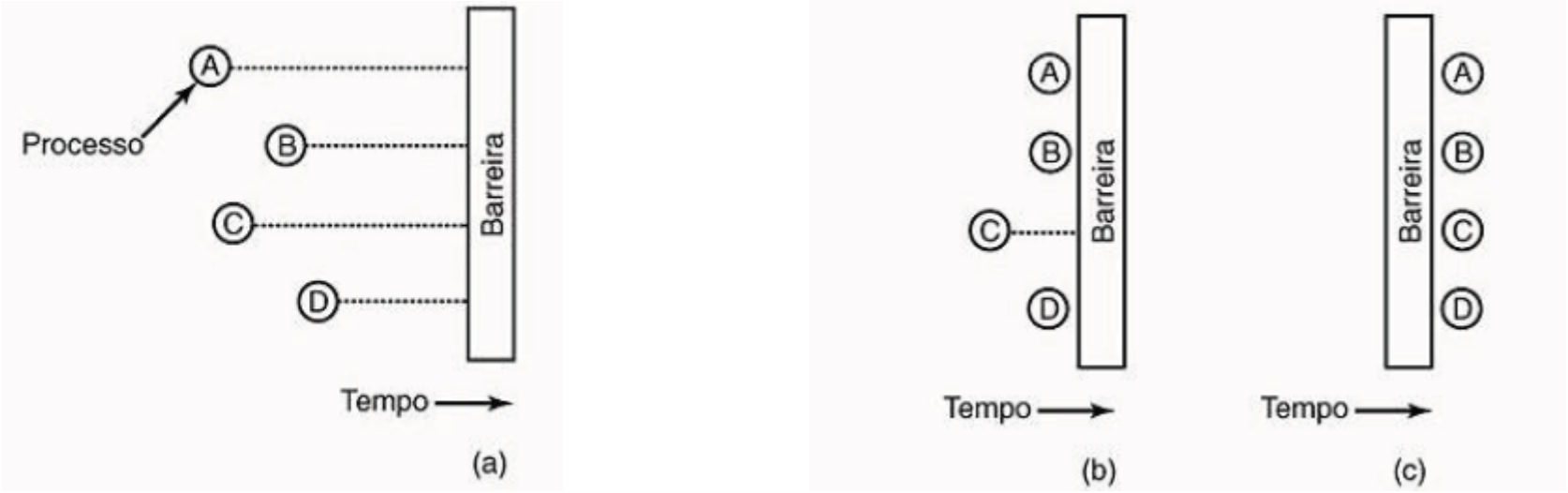


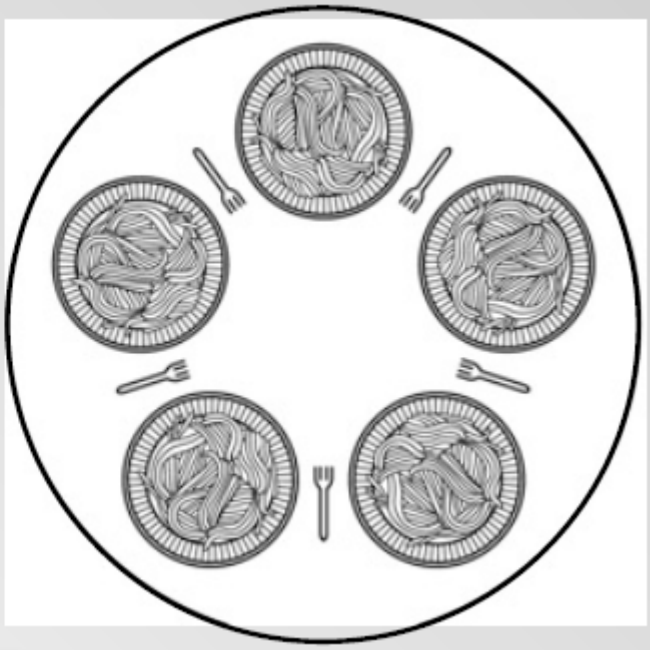
Figura 2.30 Uso de uma barreira. (a) Processos se aproximando de uma barreira. (b) Todos os processos, exceto um, estão bloqueados pela barreira. (c) Quando o último processo chega à barreira, todos passam por ela.

pthread_barrier_t - inicializa informando quantidade de threads...
int pthread_barrier_wait(pthread_barrier_t *barrier)

Jantar dos Filósofos

- Problema clássico de IPC

- Filósofos comem/pensam
- Cada um precisa de 2 garfos para comer
- Pega um garfo por vez
- Como prevenir *deadlock* ?



Jantar dos Filósofos

- Solução Óbvia

```
#define N 5                                     /* número de filósofos */

void philosopher(int i)                         /* i: número do filósofo, de 0 a 4 */
{
        while (TRUE) {
                think( );                       /* o filósofo está pensando */
                take_fork(i);                   /* pega o garfo esquerdo */
                take_fork((i+1) % N);           /* pega o garfo direito; % é o operador módulo */
                eat( );                         /* hummm! Espaguete*/
                put_fork(i);                    /* devolve o garfo esquerdo à mesa */
                put_fork((i+1) % N);            /* devolve o garfo direito à mesa */
        }
}
```

Centro de Informática UFPE

Jantar dos Filósofos

- Verificar se pode pegar o garfo
  - Se todos tiverem o mesmo intervalo
  - Tempo randômico

- Deixar "toda" a função *philosopher()* como região crítica
  - Daria certo?

```
void philosopher(int i)
{
        while (TRUE) {
                think( );
                take_fork(i);
                take_fork((i+1) % N);
                eat( );
                put_fork(i);
                put_fork((i+1) % N);
        }
}
```

Jantar dos Filósofos

- Solução

```
#define N           5           /* número de filósofos */
#define LEFT        (i+N-1)%N   /* número do vizinho à esquerda de i */
#define RIGHT       (i+1)%N     /* número do vizinho à direita de i */
#define THINKING    0           /* o filósofo está pensando */
#define HUNGRY      1           /* o filósofo está tentando pegar garfos */
#define EATING      2           /* o filósofo está comendo */
typedef int semaphore;          /* semáforos são um tipo especial de int */
int state[N];                   /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;            /* exclusão mútua para as regiões críticas */
semaphore s[N];                 /* um semáforo por filósofo */

void philosopher(int i)         /* i: o número do filósofo, de 0 a N-1 */
{
    while (TRUE) {              /* repete para sempre */
        think( );               /* o filósofo está pensando */
        take_forks(i);          /* pega dois garfos ou bloqueia */
        eat( );                 /* hummm! Espaguete! */
        put_forks(i);           /* devolve os dois garfos à mesa */
    }
}
```

Jantar dos Filósofos

- Solução

```
void take_forks(int i)                  /* i: o número do filósofo, de 0 a N–1 */
{
        down(&mutex);                   /* entra na região crítica */
        state[i] = HUNGRY;              /* registra que o filósofo está faminto */
        test(i);                        /* tenta pegar dois garfos */
        up(&mutex);                     /* sai da região crítica */
        down(&s[i]);                    /* bloqueia se os garfos não foram pegos */
}

void put_forks(i)                       /* i: o número do filósofo, de 0 a N–1 */
{
        down(&mutex);                   /* entra na região crítica */
        state[i] = THINKING;            /* o filósofo acabou de comer */
        test(LEFT);                     /* vê se o vizinho da esquerda pode comer agora */
        test(RIGHT);                    /* vê se o vizinho da direita pode comer agora */
        up(&mutex);                     /* sai da região crítica */
}

void test(i)                            /* i: o número do filósofo, de 0 a N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Jantar do filósofo

Sistemas Operacionais
Concorrência

Carlos Ferraz (cagf@cin.ufpe.br)
Jorge Cavalcanti Fonsêca (jcbf@cin.ufpe.br)