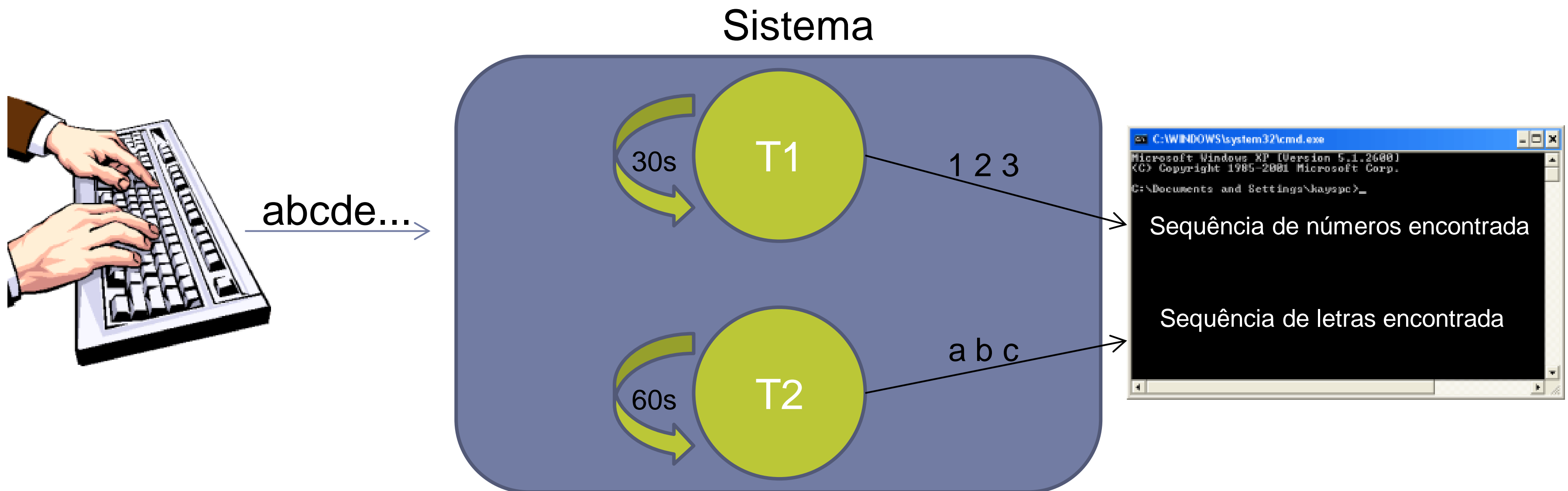
Sistemas Operacionais
Concorrência

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

Threads em Java- Exemplo



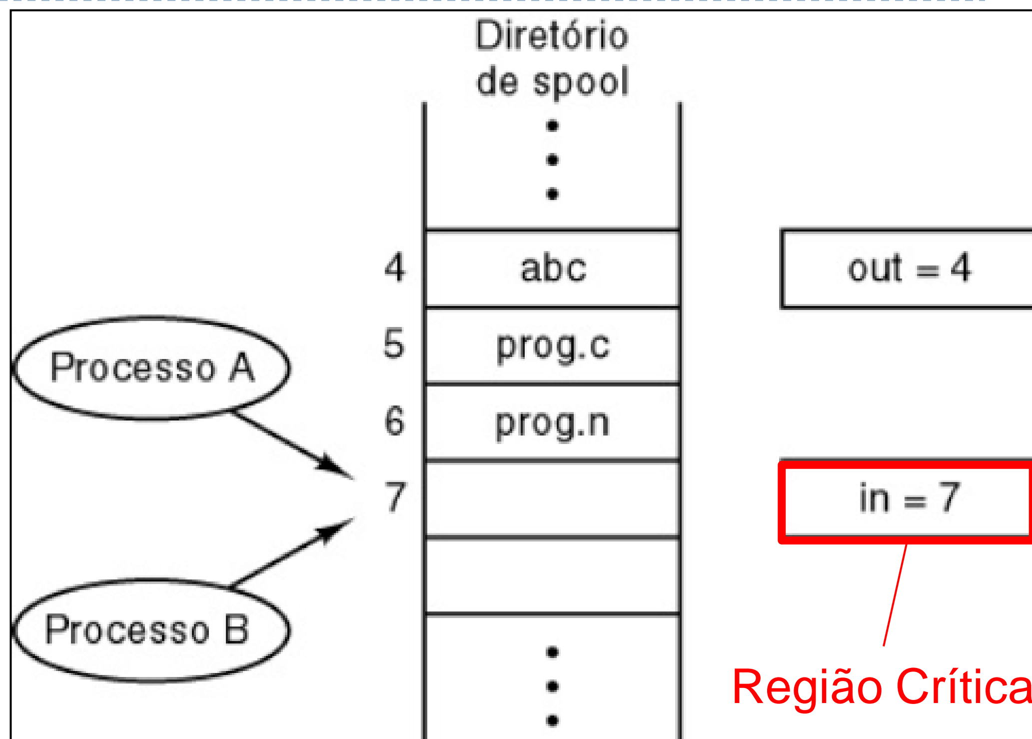
Cada thread detecta e remove a sequência encontrada
Não existe interação entre as threads
Até existe uma comunicação de informação/dados

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 que 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)

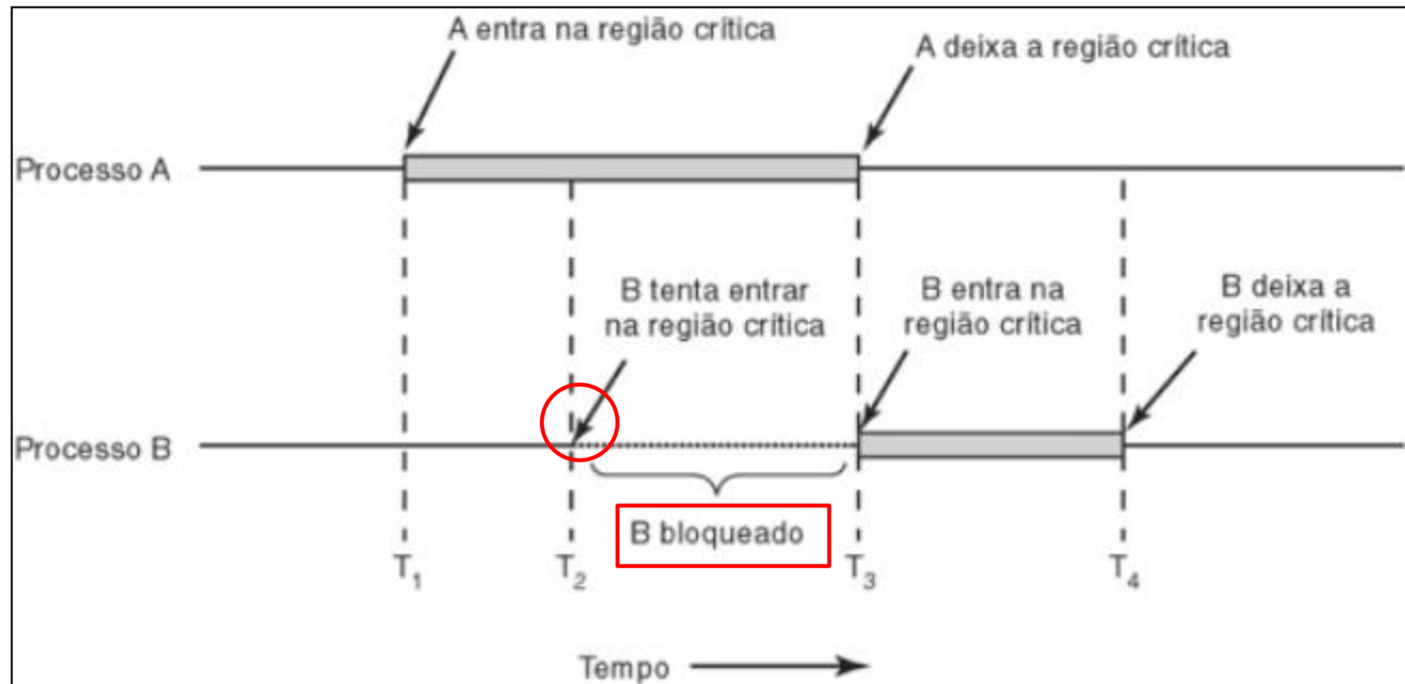
Região Crítica

- 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
  - *É 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

```
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

Exclusão Mútua

- Chaveamento obrigatório

turn = 0

```
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

Centro de Informática U-F-P-E

Exclusão Mútua

- Chaveamento obrigatório

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

Exclusão Mútua

- Chaveamento obrigatório

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

Exclusão Mútua

- Chaveamento obrigatório

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*)

Exclusão Mútua

- Chaveamento obrigatório

turn = 0
turn = 1
turn = 0

```
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*)

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*)

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*)

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 = Processa A esperando o Processo B na região NÃO crítica

Exclusão Mútua

```
#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 para exclusão mútua

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;
}
```

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;
}
```

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;
}
```

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)
}
```

Apenas 1 instrução sempre é executada por vez
Turn = 0

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

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)
}
```

Quando o processo 1 executa, deixa Turn = 1
O que vai acontecer em seguida?

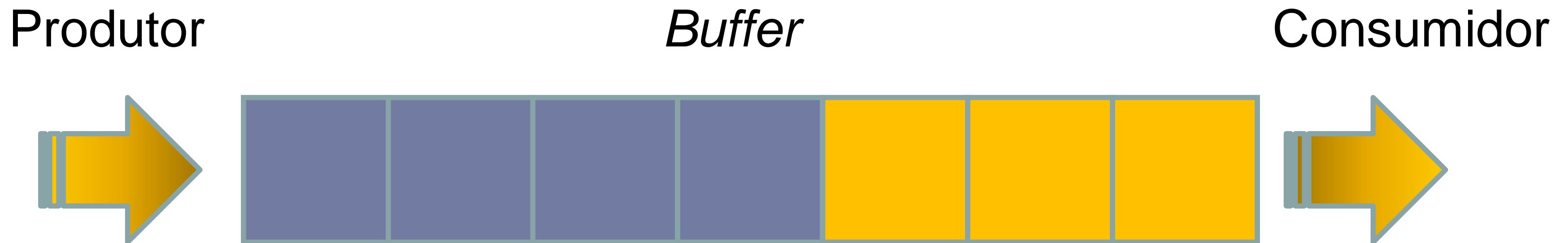
```
void leave_region(int process)
{
    interested[process] = FALSE;
}
```

Solução de Peterson funciona, mas espera ociosa continua consumindo recurso

Exclusão Mútua

- Espera ociosa
  - *Quando quer entrar em sua região critica, um processo verifica sua entrada é permitida. Se não for, ele ficará em um laço esperando ate 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? */
    }
}
```

Centro de Informática U·F·P·E

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? */
}
```

...

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 */
}
```

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? */
}
```

Consumidor fica dormindo, enquanto produtor executa...

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 */
}
```

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? */
}
```

Ao inserir um item, executa o *wakeup* no consumidor, liberando-o para continuar sua execução

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 */
}
```

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? */
}
```

Caso o produtor crie itens numa velocidade maior que o consumidor os retira, o buffer vai encher. Neste momento, o produtor vai dormir.

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 */
}
```

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? */
}
```

Com o Produtor dormindo, o consumidor consome um item, e habilita o produtor a trabalhar, acordando-o...

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 */
}
```

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 compartilhado

- 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

Produtor-Consumidor com Semáforos

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

semaphore empty = N;
semaphore full = 0 ;

void producer(void)
{
    int item;

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

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

        up(&full);
    }
}
```

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 */
semaphore mutex = 1;                       /* 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 */
            down(&mutex);                  /* entra na região crítica */
            insert_item(item);             /* põe novo item no buffer */
            up(&mutex);                    /* 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.

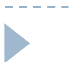
pthread_barrier_t - inicializa informando quantidade de threads...
int pthread_barrier_wait(pthread_barrier_t *barrier)

Exclusão Mútua com Pthreads

Fernando Castor e https://computing.llnl.gov/tutorials/pthreads/

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);
}

        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);
        }
```



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 | Search (String Matching)

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>

```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; 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

Um contador sequencial

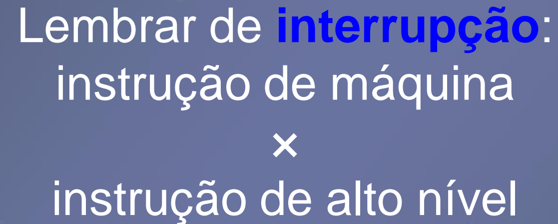
```c
#include <stdio.h>

long contador = 0;

void *inc(){
  int i = 0;
  for(; i < 9000000; i++) { contador++; }
}
void *dec(){
  int i = 0;
  for(; i < 9000000; i++) { contador--; }
}
int main (int argc, char *argv[]){
  inc();
  dec();
  printf("Valor final do contador: %ld\n", contador);
}
```

Um contador com pthreads

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

long contador = 0;

void *inc(void *threadid){
  int i = 0;
  for(; i < 9000000; i++) { contador++; }
}
void *dec(void *threadid){
  int i = 0;
  for(; i < 9000000; i++) { contador--; }
}
int main (int argc, char *argv[]){
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, inc, NULL);
  pthread_create(&thread2, NULL, dec, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  printf("Valor final do contador: %ld\n", contador);
  pthread_exit(NULL);
}
```

```
#include <stdio.h>

long contador = 0;

void *inc(){
  int i = 0;
  for(; i < 9000000; i++) { contador++; }
}
void *dec(){
  int i = 0;
  for(; i < 9000000; i++) { contador--; }
}
int main (int argc, char *argv[]){
  inc();
  dec();
  printf("Valor final do contador: %ld\n", contador);
}
```

Um contador errado com pthreads




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

long contador = 0;

void *inc(void *threadid){
  int i = 0;
  for(; i < 9000000; i++) { contador++; } // condição de corrida!
}
void *dec(void *threadid){
  int i = 0;
  for(; i < 9000000; i++) { contador--; } // condição de corrida!
}
int main (int argc, char *argv[]){
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, inc, NULL);
  pthread_create(&thread2, NULL, dec, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  printf("Valor final do contador: %ld\n", contador);
  pthread_exit(NULL);
}
```

Relembrando: exclusão mútua

- Dois processos nunca podem estar simultaneamente na mesma região crítica
- Não se pode considerar velocidades ou número de CPUs
- Nenhum processo executando fora de sua região crítica pode bloquear outros processos
- Nenhum processo deve esperar eternamente para entrar em sua região crítica

- A typical sequence in the use of a mutex is as follows:
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one succeeds and that thread owns the mutex
  - The owner thread performs some set of actions
  - The owner unlocks the mutex
  - Another thread acquires the mutex and repeats the process
  - Finally the mutex is destroyed

- When several threads compete for a mutex, the losers block at that call - an unblocking call is available with "trylock" instead of the "lock" call.
- When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so. For example, if 4 threads are updating the same data, but only one uses a mutex, the data can still be corrupted.

Exclusão mútua com pthreads

- Através do conceito de *mutex*
  - Sincronizam o acesso ao estado compartilhado
    - Independentemente do valor desse estado (=> **variáveis condicionais**)
  - Tipo especial de variável (semáforo binário)
  - Diversas funções para criar, destruir e usar *mutexes*
- Tipo de dados
  - `pthread_mutex_t`

Criação de *mutexes*

- Estática:

```
pthread_mutex_t mymutex =
        PTHREAD_MUTEX_INITIALIZER;
```

- Dinâmica:

```
pthread_mutex_t mymutex;
...
pthread_mutex_init(&mymutex,NULL);
```

Gerenciamento de *mutexes*

```
int pthread_mutex_destroy(
        pthread_mutex_t *mutex);

int pthread_mutex_init(
        pthread_mutex_t *restrict
        mutex,
        const pthread_mutexattr_t
        *restrict attr);
```

Usando *mutexes*

```
int pthread_mutex_lock(
                pthread_mutex_t *mutex);

int pthread_mutex_trylock(
                pthread_mutex_t *mutex);

int pthread_mutex_unlock(
                pthread_mutex_t *mutex);
```

- The pthread_mutex_lock() routine is used by a thread to acquire a lock on the specified *mutex* variable. If the mutex is already locked by another thread, *this call will block* the calling thread until the mutex is unlocked.
- pthread_mutex_trylock() will attempt to lock a mutex. However, if the mutex is already locked, *the routine will return immediately with a "busy" error code*. This routine may be useful in *preventing deadlock conditions*, as in a priority-inversion situation.

Um contador certo com pthreads

```c
#include <pthread.h>
#include <stdio.h>
long contador = 0;
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
void *inc(void *threadid){
  int i = 0; for(; i < 9000000; i++) {
    pthread_mutex_lock(&mymutex);
    contador++;
    pthread_mutex_unlock(&mymutex); }
}
void *dec(void *threadid){
  int i = 0;
  for(; i < 9000000; i++) {
    pthread_mutex_lock(&mymutex);
    contador--;
    pthread_mutex_unlock(&mymutex); }
}
int main (int argc, char *argv[]){
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, inc, NULL);
  pthread_create(&thread2, NULL, dec, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  printf("Valor final do contador: %ld\n", contador);
  pthread_exit(NULL);
}
```

```c
#include <pthread.h>
#include <stdio.h>
long contador = 0;

void *inc(void *threadid){
  int i = 0;
  for(; i < 9000000; i++) {
    contador++; // condição de corrid
  }
}
void *dec(void *threadid){
  int i = 0;
  for(; i < 9000000; i++) {

    contador--; // condição de corrid
  }
}
int main (int argc, char *argv[]){
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, inc,
  pthread_create(&thread2, NULL, dec,
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  printf("Valor final do contador: %l
    contador);
  pthread_exit(NULL);
}
```

There is nothing "magical" about mutexes...in fact they are akin to a "gentlemen's agreement" between participating threads. It is up to the code writer to insure that the necessary threads all make the mutex lock and unlock calls correctly. The following scenario demonstrates a logical error:

Thread 1	**Thread 2**	**Thread 3**
Lock	Lock	
A = 2	A = A+1	A = A*B
Unlock	Unlock	

Centro de Informática U·F·P·E

Question

- Question: When more than one thread is waiting for a locked mutex, which thread will be granted the lock first after it is released?
- ANSWER: Unless thread priority scheduling (not covered) is used, the assignment will be left to the native system scheduler and may appear to be more or less random.

Variáveis Condicionais com Pthreads

Fernando Castor e https://computing.llnl.gov/tutorials/pthreads/

Sistemas Operacionais
Concorrência

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