QXD0013 - Sistemas Operacionais

Comunicação entre Processos

Thiago Werlley Bandeira da Silva[1]

[1]Universidade Federal do Ceará, Brazil

03/11/2021

Introdução

- Processos quase sempre precisam **comunicar-se** com **outros processos**
- **Por exemplo**: em um pipeline do interpretador de comandos, a saída do primeiro processo tem de ser passada para o segundo, e assim por diante até o fim da linha.
- Sigla de comunicação entre processos: **IPC** (*interprocess communication*)
- Três tópicos importantes sobre IPC:
  - Troca de informações entre processos
  - Prevenção de conflitos entre processos
  - Sequenciamento de execução (dependências) entre processos
- Esses **três tópicos** são igualmente aplicáveis às threads

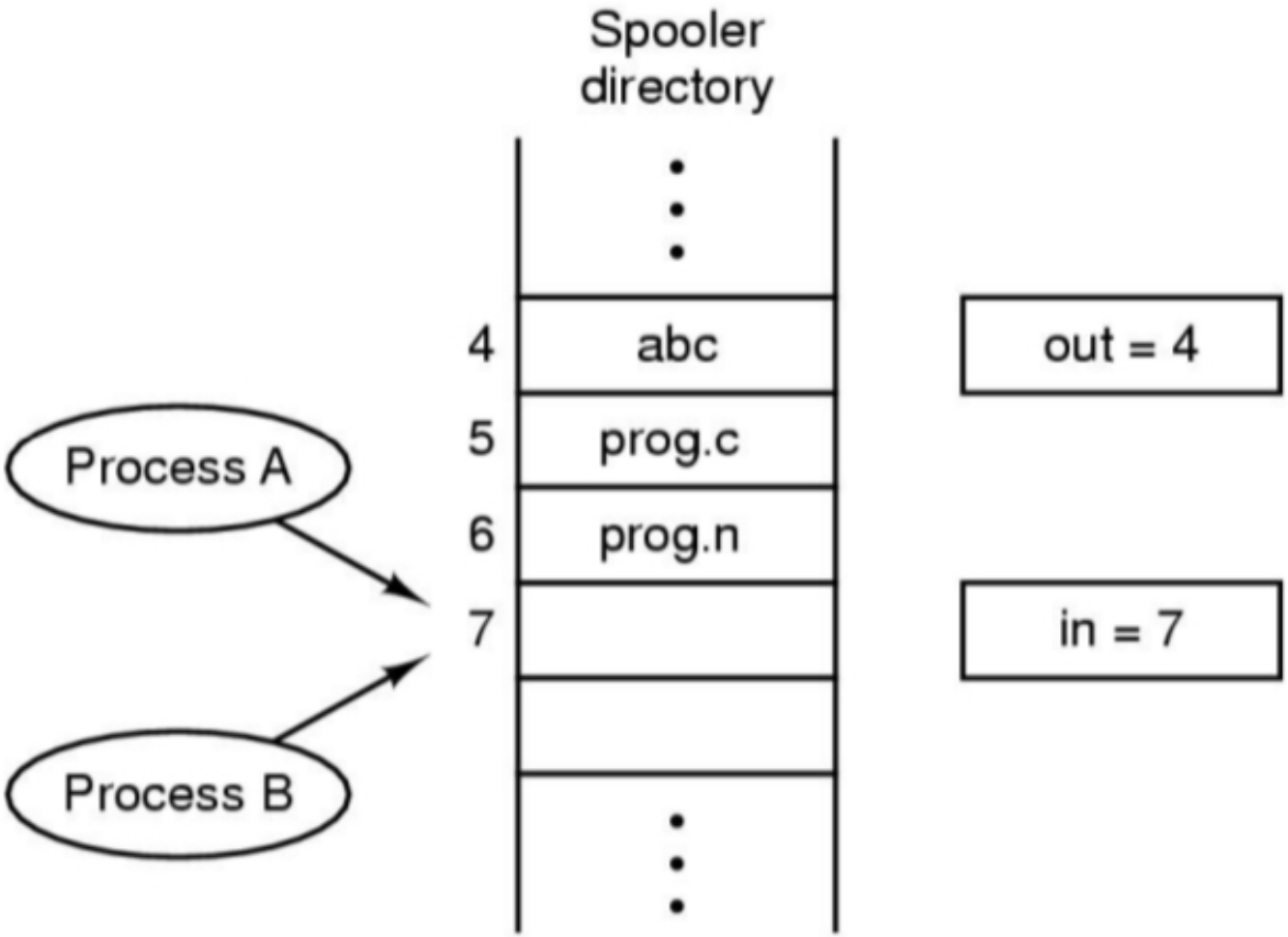
Condições de Corrida

- Em alguns sistemas operacionais, **processos** que estão **trabalhando juntos** podem **compartilhar** de alguma memória comum que **cada um pode ler e escrever**.
- A **memória compartilhada** pode encontrar-se na:
  - Memória principal
  - Arquivo compartilhado
- O local da memória compartilhada **não muda a natureza** da **comunicação** ou os **problemas** que surgem
- Situações em que dois ou mais processos estão lendo ou escrevendo alguns dados compartilhados e o resultado final depende de quem executa precisamente e quando, são chamadas de **condições de corrida**.
- Problemas ocorrem de maneira aleatória
- Difícil depuração

Exemplo: Spool de impressão

- Processo insere nome de arquivo no diretório de spool
- Daemons de impressão lê periodicamente spool
- Vagas numeradas no spool
- Variável saída: próxima impressão (*out*)
- Variável entrada: próxima vaga (*in*)
- Vagas 0 a 3 vazias e 4 a 6 ocupadas
- in $= 7$ / out $= 4$
- Processos A e B desejam imprimir

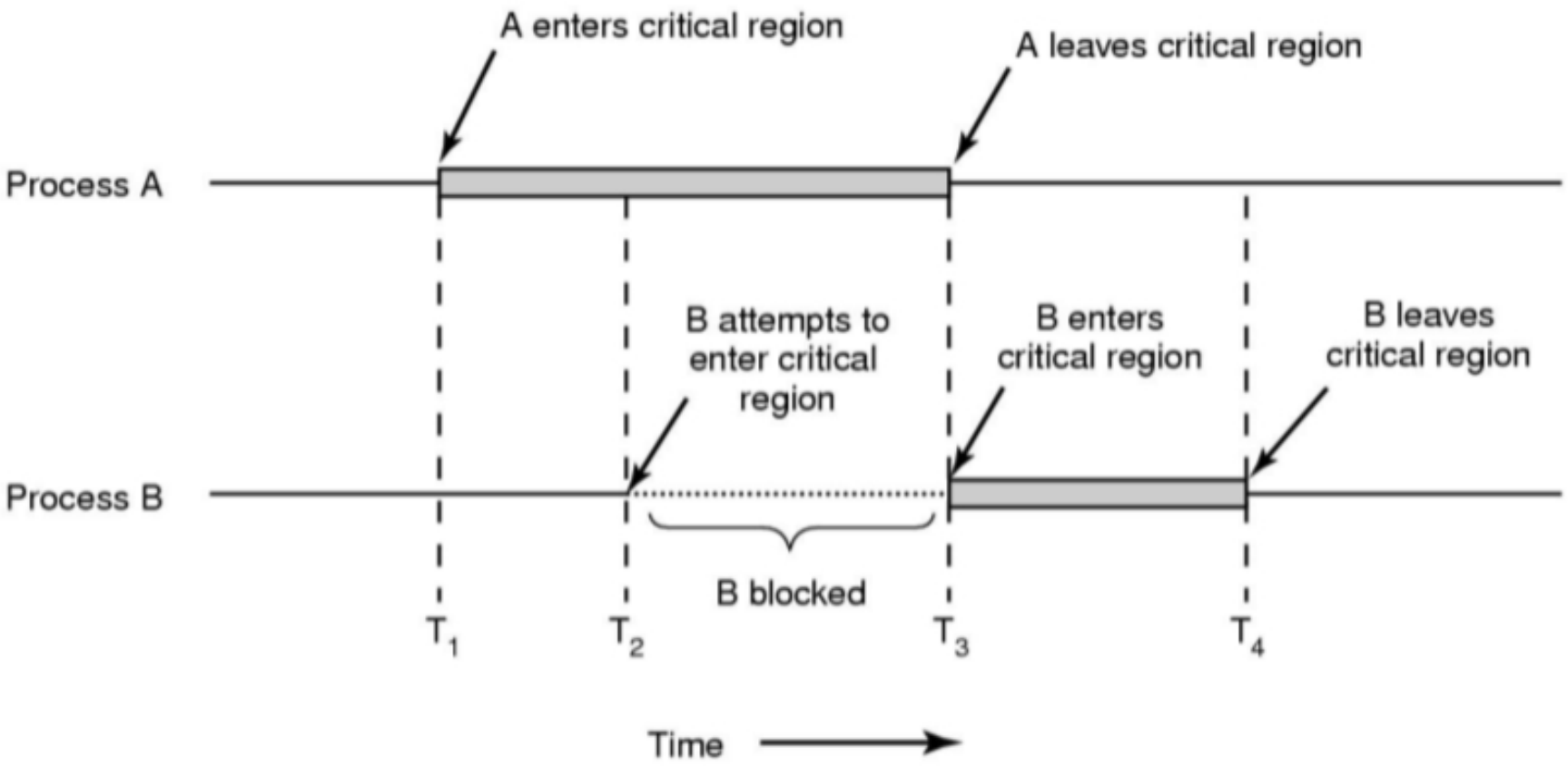
Exemplo: Spool de impressão



Regiões Críticas

- **Como evitar as condições de corrida?**
- Modo de impedir acesso simultâneo à memória compartilhada – **é encontrar alguma maneira de proibir mais de um processo de ler e escrever os dados compartilhados ao mesmo tempo**
- Exclusão mútua (*Mutual Exclusion*)
- Do exemplo: B acessa variável antes de A terminar
- Acesso a dados compartilhados $\rightarrow$ **região crítica**
- Quatro condições devem ser satisfeitas:
  - Nunca dois processos na região crítica simultaneamente
  - Nenhuma premissa sobre velocidade ou número de CPUs
  - Nenhum processo fora da região crítica pode bloquear outro
  - Nenhum processo deve esperar eternamente para entrar na região crítica

Regiões Críticas



Exclusão Mútua - Desabilitando Interrupções

- CPU chaveia entre processos mediante interrupção
- Solução mais simples:
  - Desabilitar interrupções ao adentrar em uma região crítica
- Problemas:
  - Muito poder a um processo
  - Não funciona em sistemas multiprocessados
- Comum ao SO em alguns pontos específicos (desuso)
- Inadequada como mecanismo geral

Exclusão Mútua - Variáveis de trava

- Como uma segunda tentativa, vamos procurar por uma **solução de software**
- Única variável trava (*lock*)
  - trava $= 0 \rightarrow$ região crítica livre
  - trava $= 1 \rightarrow$ região crítica ocupada
- Inicialização com valor igual a 0
- Antes de adentrar na região crítica, processo verifica trava
- Caso igual a 0, valor modificado para 1 e processo na região crítica
- Caso igual a 1, processo aguarda até tornar-se 0
- Não resolve a condição de corrida
  - Processo interrompido logo após verificar trava

Exclusão Mútua - Chaveamento Obrigatório

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

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

- Testar continuamente uma variável até que algum valor apareça é chamado de **espera ocupada**
- Alternância obrigatória entre processos
- Eficaz na prevenção de condições de corrida
- Problemas:
  - Espera ocupada consome CPU
  - Inadequado quando um processo mais lento
- Viola: *Nenhum processo fora da região crítica pode bloquear outro*

Exclusão Mútua - Solução de Peterson

- Desenvolvido em 1981 por Gary L. Peterson
- Combinação de **alternância** com **trava**
- Consiste em:
  - Duas rotinas: *enter_region* e *leave_region*
  - Uma trava
  - Um vetor de interesses
- *enter_region* é chamada antes do acesso à região crítica
- *leave_region* é chamada após saída da região crítica

Exclusão Mútua - Solução de Peterson

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Exclusão Mútua - Solução de Peterson

```
#define FALSE  0    Processo 0 executanto
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

interested[0] = 0

interested[1] = 0

other = 1

Exclusão Mútua - Solução de Peterson

```
#define FALSE  0    Processo 0 executanto
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

interested[0] = 1

interested[1] = 0

other = 1

Exclusão Mútua - Solução de Peterson

```
#define FALSE  0    Processo 0 executanto
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

interested[0] = 1

interested[1] = 0

other = 1

turn = 0

Exclusão Mútua - Solução de Peterson

```
#define FALSE  0    Processo 0 executanto
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

interested[0] = 1

interested[1] = 0

other = 1

turn = 0

Exclusão Mútua - Solução de Peterson

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Processo 0 executanto

interested[0] = 1

interested[1] = 0

other = 1

turn = 0

Processo 1 passa a executar

Exclusão Mútua - Solução de Peterson

```
#define FALSE  0    Processo 1 executanto
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

interested[0] = 1

interested[1] = 0

other = 0

Exclusão Mútua - Solução de Peterson

```
#define FALSE  0    Processo 1 executanto
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

interested[0] = 1

interested[1] = 1

other = 0

Exclusão Mútua - Solução de Peterson

```
#define FALSE  0    Processo 1 executanto
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

interested[0] = 1

interested[1] = 1

other = 0

turn = 1

Exclusão Mútua - Instrução TSL

- Instrução assembly: *Test and Set Lock*
  - TSL RX,LOCK
  - Escreve o conteúdo de LOCK em RX e escreve valor 1 em LOCK
  - Quando LOCK $\neq 0$, acesso ao barramento de memória proibido
- Operações de leitura e armazenamento indivisíveis
- Efetivo em plataformas multiprocessadas

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was nonzero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

Exclusão Mútua - Instrução TSL

- Instrução assembly: *Test and Set Lock*
  - TSL RX,LOCK
  - Escreve o conteúdo de LOCK em RX e escreve valor 1 em LOCK
  - Quando LOCK $\neq 0$, acesso ao barramento de memória proibido
- Operações de leitura e armazenamento indivisíveis
- Efetivo em plataformas multiprocessadas

```
enter_region:
    TSL REGISTER,LOCK    | copy lock to register and set lock to 1
    CMP REGISTER,#0      | was lock zero?
    JNE enter_region     | if it was nonzero, lock was set, so loop
    RET                  | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0         | store a 0 in lock
    RET                  | return to caller
```

Tentativa de entrar na região crítica quando já ocupada

LOCK = 1

REGISTER = 1

Exclusão Mútua - Instrução TSL

- Instrução assembly: *Test and Set Lock*
  - TSL RX,LOCK
  - Escreve o conteúdo de LOCK em RX e escreve valor 1 em LOCK
  - Quando LOCK $\neq 0$, acesso ao barramento de memória proibido
- Operações de leitura e armazenamento indivisíveis
- Efetivo em plataformas multiprocessadas

```
enter_region:
      TSL REGISTER,LOCK     | copy lock to register and set lock to 1
      CMP REGISTER,#0       | was lock zero?
      JNE enter_region      | if it was nonzero, lock was set, so loop
      RET                   | return to caller; critical region entered

leave_region:
      MOVE LOCK,#0          | store a 0 in lock
      RET                   | return to caller
```

Tentativa de entrar na região crítica quando já ocupada

LOCK = 1

REGISTER = 1

Exclusão Mútua - Instrução TSL

- Instrução assembly: *Test and Set Lock*
  - TSL RX,LOCK
  - Escreve o conteúdo de LOCK em RX e escreve valor 1 em LOCK
  - Quando LOCK $\neq 0$, acesso ao barramento de memória proibido
- Operações de leitura e armazenamento indivisíveis
- Efetivo em plataformas multiprocessadas

```
enter_region:
        TSL REGISTER,LOCK     | copy lock to register and set lock to 1
        CMP REGISTER,#0       | was lock zero?
        JNE enter_region      | if it was nonzero, lock was set, so loop
        RET                   | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0          | store a 0 in lock
        RET                   | return to caller
```

Tentativa de entrar na região crítica quando já ocupada

LOCK = 1

REGISTER = 1

Exclusão Mútua - Instrução TSL

- Instrução assembly: *Test and Set Lock*
  - TSL RX,LOCK
  - Escreve o conteúdo de LOCK em RX e escreve valor 1 em LOCK
  - Quando LOCK $\neq 0$, acesso ao barramento de memória proibido
- Operações de leitura e armazenamento indivisíveis
- Efetivo em plataformas multiprocessadas

```
enter_region:
    TSL REGISTER,LOCK   | copy lock to register and set lock to 1
    CMP REGISTER,#0     | was lock zero?
    JNE enter_region    | if it was nonzero, lock was set, so loop
    RET                 | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0        | store a 0 in lock
    RET                 | return to caller
```

Tentativa de entrar na região crítica quando já ocupada

LOCK = 1

REGISTER = 1

Exclusão Mútua - Instrução TSL

- Instrução assembly: *Test and Set Lock*
  - TSL RX,LOCK
  - Escreve o conteúdo de LOCK em RX e escreve valor 1 em LOCK
  - Quando LOCK $\neq 0$, acesso ao barramento de memória proibido
- Operações de leitura e armazenamento indivisíveis
- Efetivo em plataformas multiprocessadas

```
enter_region:
    TSL REGISTER,LOCK   | copy lock to register and set lock to 1
    CMP REGISTER,#0     | was lock zero?
    JNE enter_region    | if it was nonzero, lock was set, so loop
    RET                 | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0        | store a 0 in lock
    RET                 | return to caller
```

Tentativa de entrar na região crítica liberada

LOCK = 0

REGISTER = 1

Exclusão Mútua - Instrução TSL

- Instrução assembly: *Test and Set Lock*
  - TSL RX,LOCK
  - Escreve o conteúdo de LOCK em RX e escreve valor 1 em LOCK
  - Quando LOCK $\neq 0$, acesso ao barramento de memória proibido
- Operações de leitura e armazenamento indivisíveis
- Efetivo em plataformas multiprocessadas

```
enter_region:
        TSL REGISTER,LOCK     | copy lock to register and set lock to 1
        CMP REGISTER,#0       | was lock zero?
        JNE enter_region      | if it was nonzero, lock was set, so loop
        RET                   | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0          | store a 0 in lock
        RET                   | return to caller
```

Tentativa de entrar na região crítica liberada

LOCK = 0

REGISTER = 1

Exclusão Mútua - Instrução TSL

- Instrução assembly: *Test and Set Lock*
  - TSL RX,LOCK
  - Escreve o conteúdo de LOCK em RX e escreve valor 1 em LOCK
  - Quando LOCK $\neq 0$, acesso ao barramento de memória proibido
- Operações de leitura e armazenamento indivisíveis
- Efetivo em plataformas multiprocessadas

```
enter_region:
    TSL REGISTER,LOCK   | copy lock to register and set lock to 1
    CMP REGISTER,#0     | was lock zero?
    JNE enter_region    | if it was nonzero, lock was set, so loop
    RET                 | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0        | store a 0 in lock
    RET                 | return to caller
```

Tentativa de entrar na região crítica liberada

LOCK = 1

REGISTER = 0

Exclusão Mútua - Instrução TSL

- Instrução assembly: *Test and Set Lock*
  - TSL RX,LOCK
  - Escreve o conteúdo de LOCK em RX e escreve valor 1 em LOCK
  - Quando LOCK $\neq 0$, acesso ao barramento de memória proibido
- Operações de leitura e armazenamento indivisíveis
- Efetivo em plataformas multiprocessadas

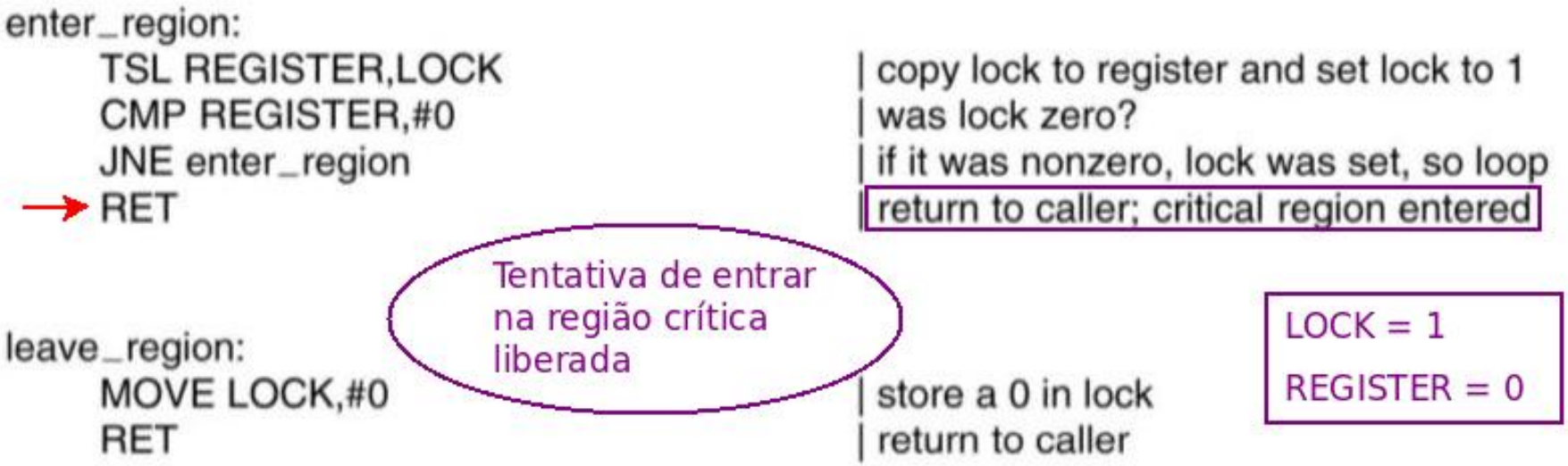
```
enter_region:
    TSL REGISTER,LOCK     | copy lock to register and set lock to 1
    CMP REGISTER,#0       | was lock zero?
    JNE enter_region      | if it was nonzero, lock was set, so loop
    RET                   | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0          | store a 0 in lock
    RET                   | return to caller
```

Tentativa de entrar na região crítica liberada

LOCK = 1
REGISTER = 0

Exclusão Mútua - Instrução TSL

- Instrução assembly: *Test and Set Lock*
  - TSL RX,LOCK
  - Escreve o conteúdo de LOCK em RX e escreve valor 1 em LOCK
  - Quando LOCK $\neq 0$, acesso ao barramento de memória proibido
- Operações de leitura e armazenamento indivisíveis
- Efetivo em plataformas multiprocessadas



Exclusão Mútua - Dormir e Acordar

- Peterson, TSL, XCHG $\rightarrow$ espera ocupada (ociosa)
  - Laço de espera até que condição seja satisfeita
  - Gasta tempo de CPU
  - Efeitos inesperados: **inversão de prioridade**
- Exemplo:
  - Dois processos: H (alta) e L (baixa) prioridades
  - H deve ser executado sempre que pronto
  - Se H torna-se pronto e L na região crítica
  - H em laço infinito e L não deixa região crítica
  - Acontece o que chamamos de problema de **inversão de prioridade**
- Primitivas de comunicação entre processos que bloqueiam em vez de desperdiçar tempo da CPU:
  - sleep: processo é bloqueado
  - wakeup: processo torna-se pronto

Exclusão Mútua - Dormir e Acordar

- Exemplo: Problema produtor-consumidor (também conhecido como problema do **buffer limitado**)
  - Buffer compartilhado por dois processos
  - Produtor: insere informação no buffer
  - Consumidor: retira informação do buffer
- O **problema** surge quando o produtor quer **colocar um item novo no buffer**, mas ele já está **cheio**. A solução é o **produtor ir dormir**, para ser desperto quando o **consumidor** tiver **removido** um ou mais itens. De modo similar, se o consumidor quer **remover um item do buffer** e vê que este está **vazio**, ele **vai dormir** até o **produtor** colocar algo no **buffer e despertá-lo**.

Exclusão Mútua - Dormir e Acordar

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Exclusão Mútua - Dormir e Acordar

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

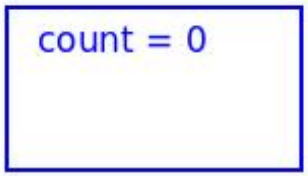
Exclusão Mútua - Dormir e Acordar

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

count = 0

Exclusão Mútua - Dormir e Acordar

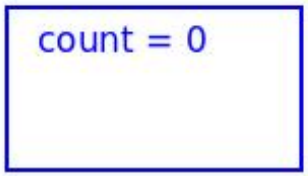
```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```



Interrupção

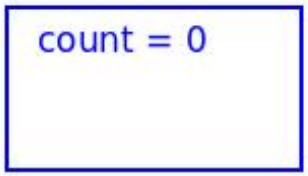
Exclusão Mútua - Dormir e Acordar

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```



Exclusão Mútua - Dormir e Acordar

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```



Exclusão Mútua - Dormir e Acordar

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```


count = 0

Exclusão Mútua - Dormir e Acordar

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Exclusão Mútua - Dormir e Acordar

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

count = 1

Exclusão Mútua - Dormir e Acordar

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

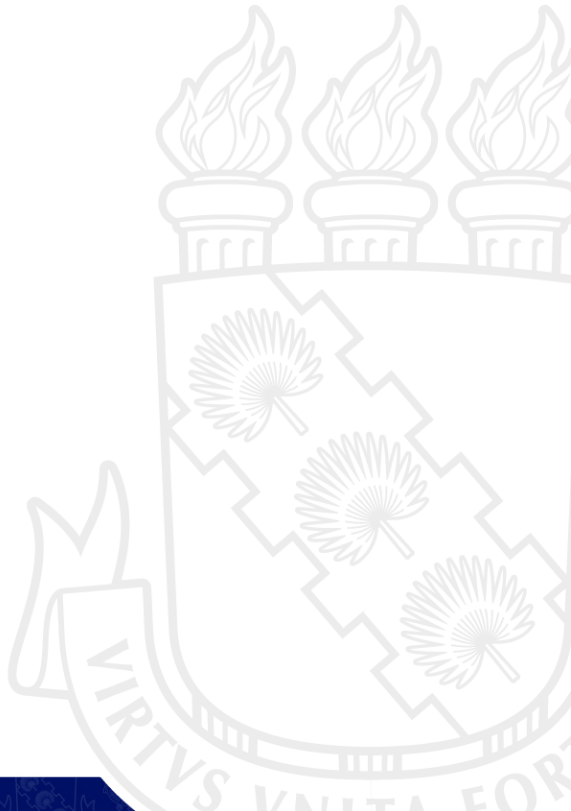
Retorno Interrupção

Exclusão Mútua - Semáforos

- E.W. Dijkstra: sugeriu usar uma variável inteira para contar o número de sinais de acordar salvos para uso futuro
- Semáforo
  - Valor 0: nenhum sinal de acordar salvo
  - Valor N: N sinais de acordar pendentes
- Duas operações: down e up (generalizações de sleep e wakeup)
- down:
  - Verifica se valor é maior do que zero
  - Caso verdadeiro, decrementa o valor
  - Caso falso, o processo é posto para dormir
- up:
  - Incrementa o valor do semáforo
  - Verifica quais processos estão dormindo no semáforo
  - Escolhe um para ser acordado

Exclusão Mútua - Semáforos

- Operações indivisíveis
- Chamadas de sistema:
  - Poucas instruções
  - Desabilitação de interrupções
- Sistemas com mais de uma CPU:
  - TSL/XCHG para proteger semáforo



Exclusão Mútua - Semáforos

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

Exclusão Mútua - Semáforos

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item( );                 /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item( );                  /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

3 semáforos

Exclusão Mútua - Semáforos

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                          /* decrement empty count */
          down(&mutex);                          /* enter critical region */
          insert_item(item);                     /* put new item in buffer */
          up(&mutex);                            /* leave critical region */
          up(&full);                             /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                           /* decrement full count */
          down(&mutex);                          /* enter critical region */
          item = remove_item( );                 /* take item from buffer */
          up(&mutex);                            /* leave critical region */
          up(&empty);                            /* increment count of empty slots */
          consume_item(item);                    /* do something with the item */
     }
}
```

Lugares não preenchidos

Exclusão Mútua - Semáforos

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item( );                 /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item( );                  /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

Lugares vagos

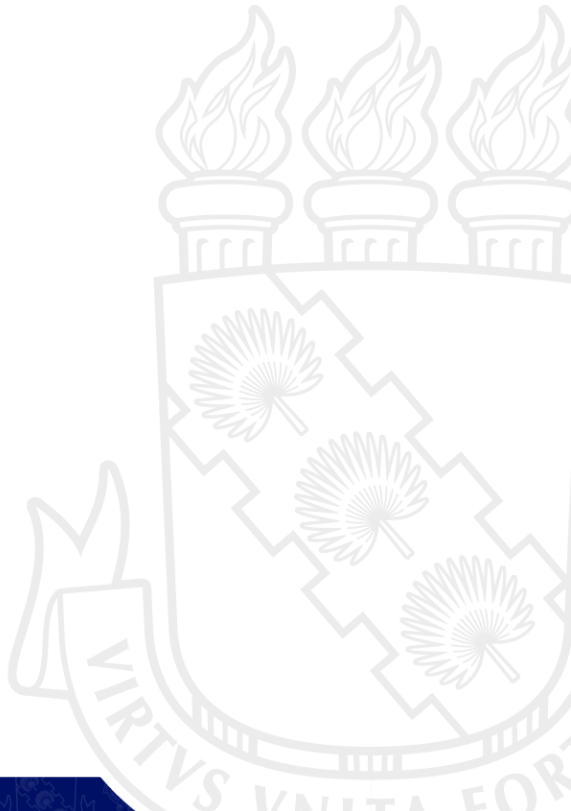
Exclusão Mútua - Semáforos

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item( );                 /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item( );                  /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

Impedir que produtor e consumidor acessem o buffer simultaneamente

Exclusão Mútua - Semáforos

- Semáforos de duas maneiras:
  - mutex: exclusão mútua
  - full e empty: sincronização
- Outro exemplo:
  - Um semáforo para cada dispositivo E/S
  - Após inicialização, down realizado
  - Rotina associada à interrupção realiza up
  - Processo que gerencia dispositivo pronto



Exclusão Mútua - Mutexes

- Abreviação de *Mutual Exclusion*
- Semáforos binários
- Exclusão Mútua
  - Recurso
  - Código compartilhado
- Dois estados: impedido/desimpedido
- Simplicidade $\rightarrow$ TSL/XCHG