Sistemas Operacionais

Processos - Parte 2

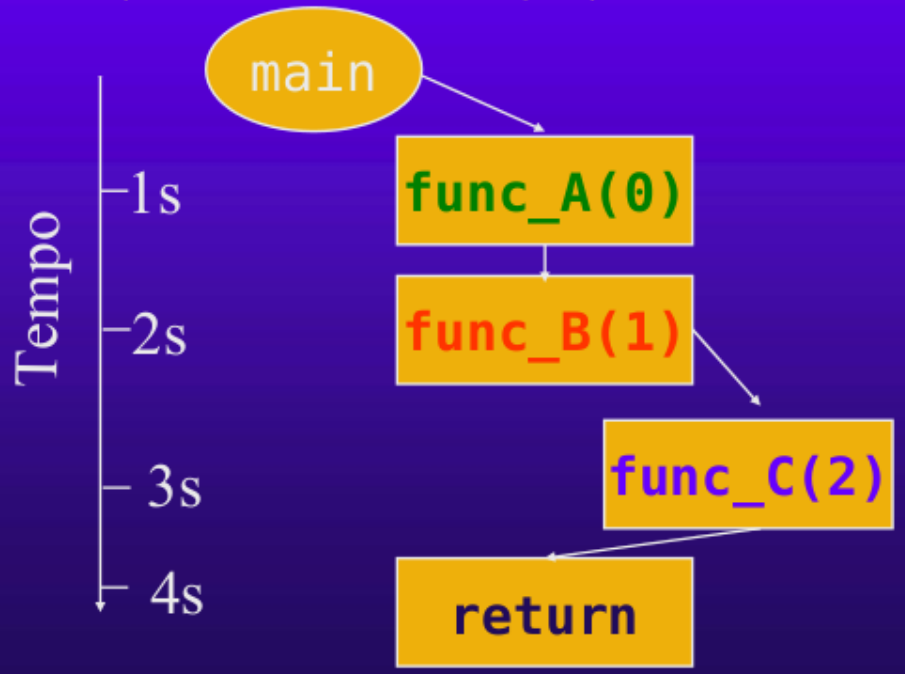
Prof. Dr. Fábio Rodrigues de la Rocha

UTFPR

Conceitos de programação concorrente

- **Programa ordinário:**
  - declaração de dados
  - descrição de instruções executáveis
  - execução seqüencial

  (um fluxo de execução)

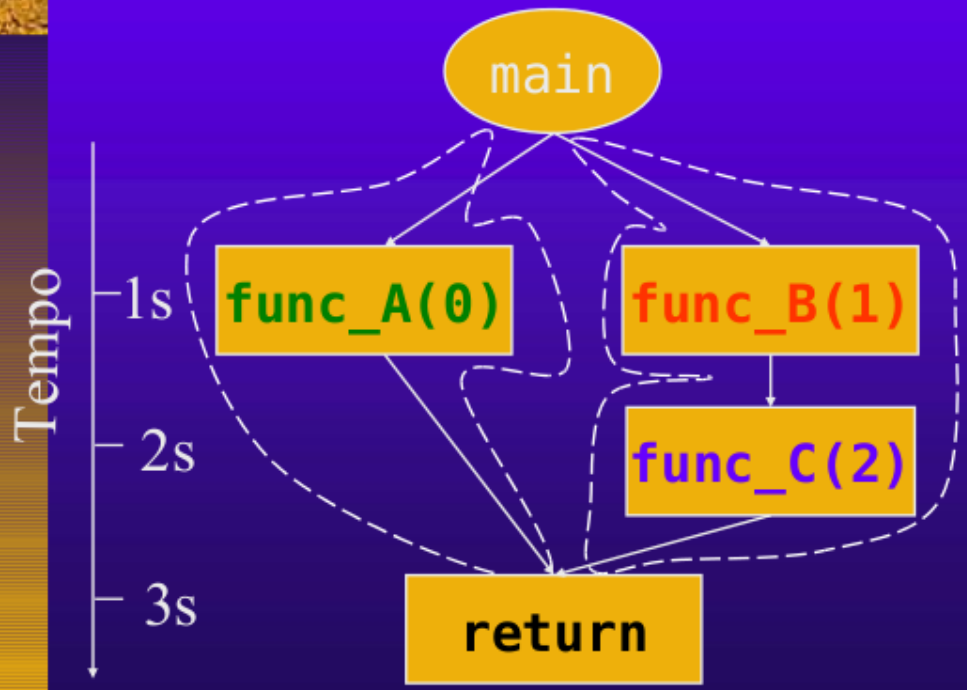


Programa ordi.c

```
 1: #include <stdio.h>
 2: int x = 0; // declaração
 3:
 4: void func_A(int v)
 5: {
 6:       printf("A %d\n",v);
 7: }
 8: void func_B(int v)
 9: {
10:       printf("B %d\n",v);
11:       func_C(v+1);
12: }
13: void func_C(int v)
14: {
15:       printf("C %d\n",v);
16: }
17: // Programa principal
18: int main (void)
19: {
20:       func_A(x);
21:       func_B(x+1);
22:       return 0;
23: }
24:
```

Conceitos de programação concorrente

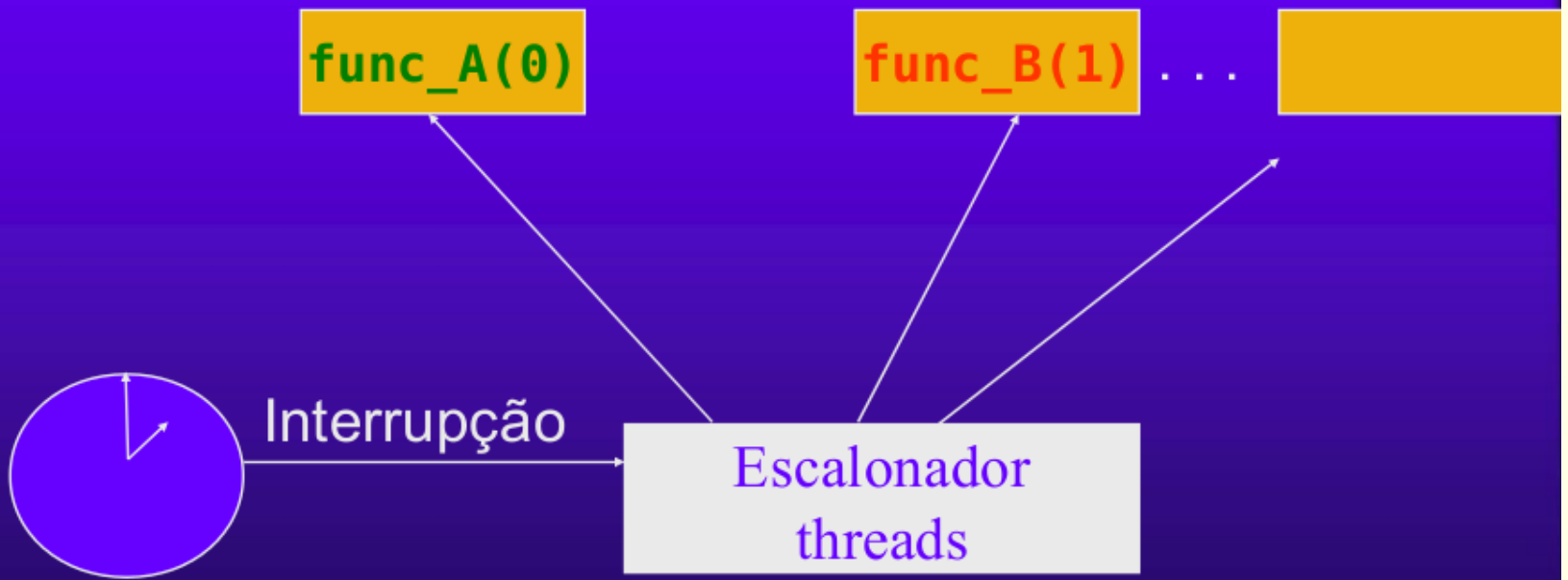
- **Programa concorrente:**
  - conjunto de programas ordinários
  - execução paralela (mesmo que abstrata)
  - Processo-thread ⇨ programa ordinário
  - Prog. concorrente ⇨ conjunto de threads



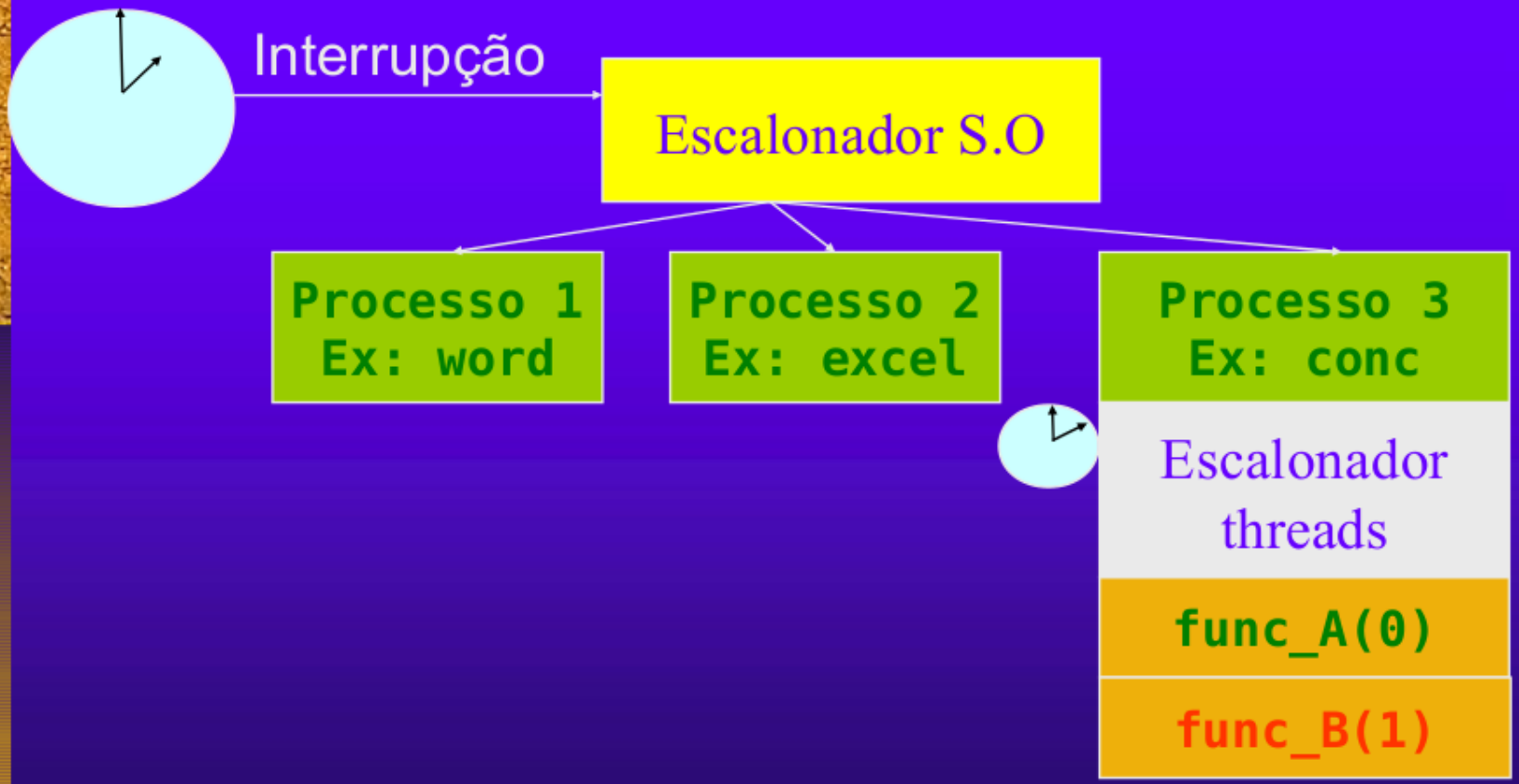
Programa conc.c

```
 1: #include <stdio.h>
 2: int x = 0; // declaração
 3:
 4: void func_A(int v)
 5: {
 6:       printf("A %d\n",v);
 7: }
 8: void func_B(int v)
 9: {
10:       printf("B %d\n",v);
11:       func_C(v+1);
12: }
13: void func_C(int v)
14: {
15:       printf("C %d\n",v);
16: }
17: // Programa principal
18: int main (void)
19: {
CreateThread(func_A(x),
20:       func_B(x+1));
21:       return 0;
22: }
23:
```

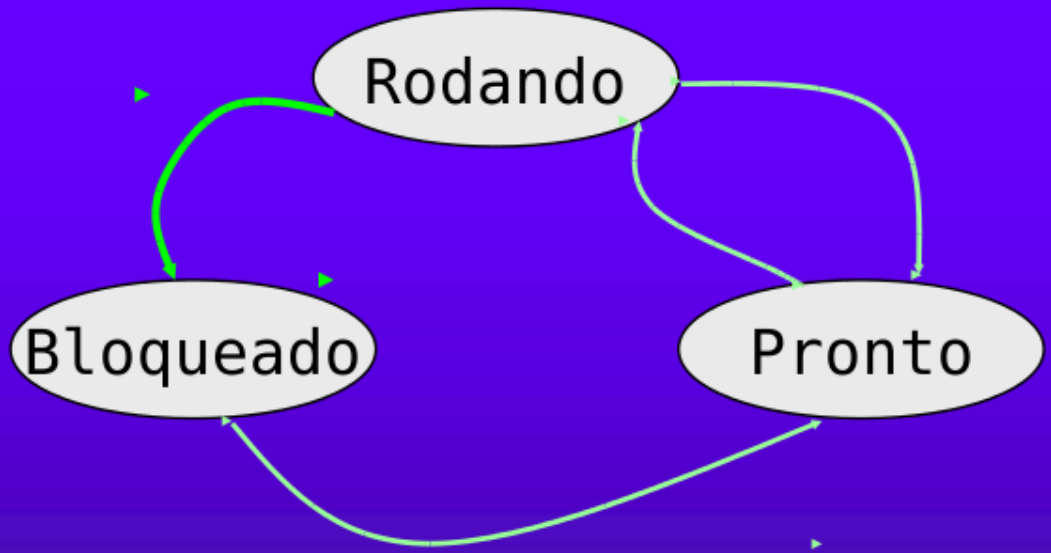
Execução “paralela”

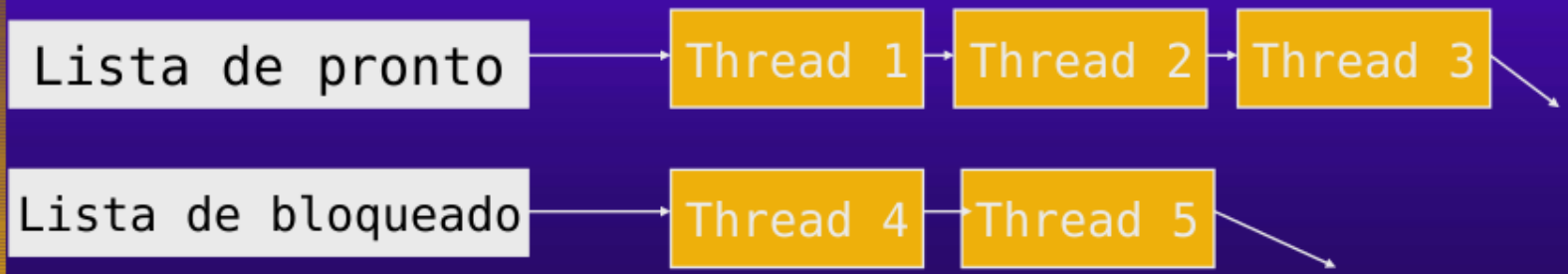


Escalonamento de processos e escalonamento de threads



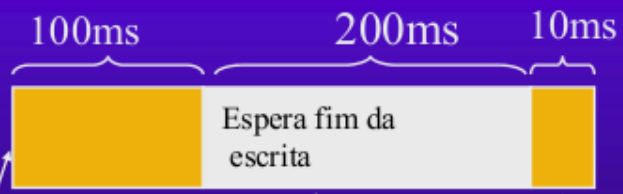
Estados dos processos ou threads





Vantagens da programação concorrente sobre a convencional

A
- valor=resolve_sistema_equações();
- write(valor,10); escreve na posição de memória 10
- Valor=(valor+1)*3



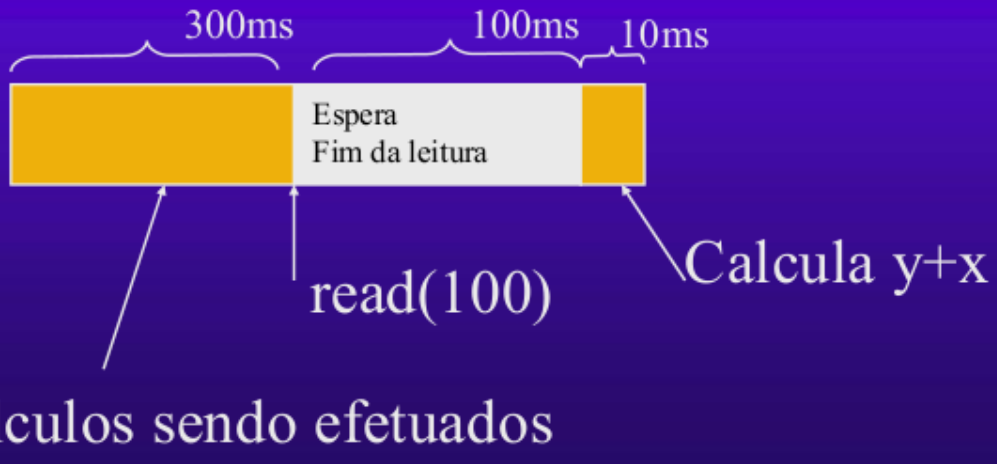
Resolução do sistema de equações

write

Calcula (valor+1)*3

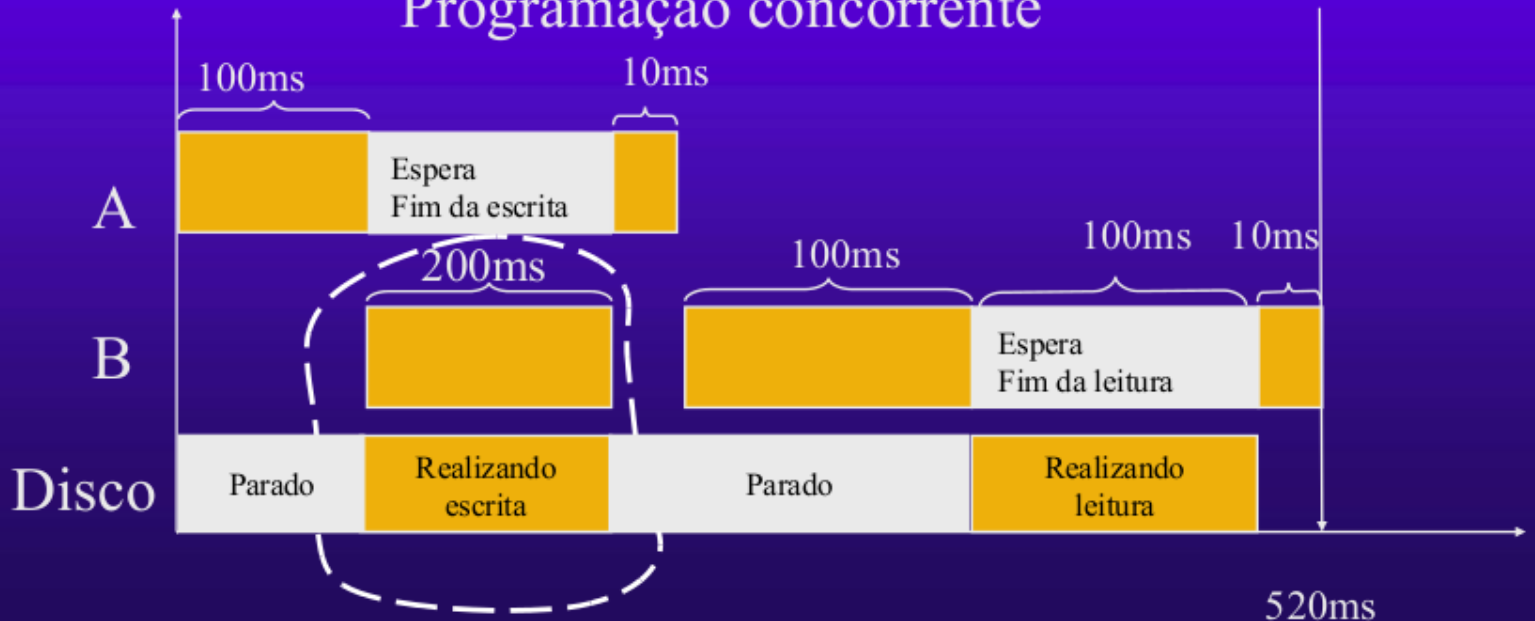
Vantagens da programação concorrente sobre a convencional

B {
- Y=Realiza_calculos()
- X=read(100); Lê a posicao 100 da memória
- Z=Y+X;



Programação convencional

Programação concorrente



Desvantagens da programação concorrente

- Está sujeita a todos os erros da programação convencional
- Mais todos os problemas que surgem quando existe comunicação entre processos
  - Comunicação no sentido de troca de mensagens
  - Mas também no sentido de acesso à áreas de memória ou estruturas de dados compartilhadas

Comunicação entre processos - Semáforos

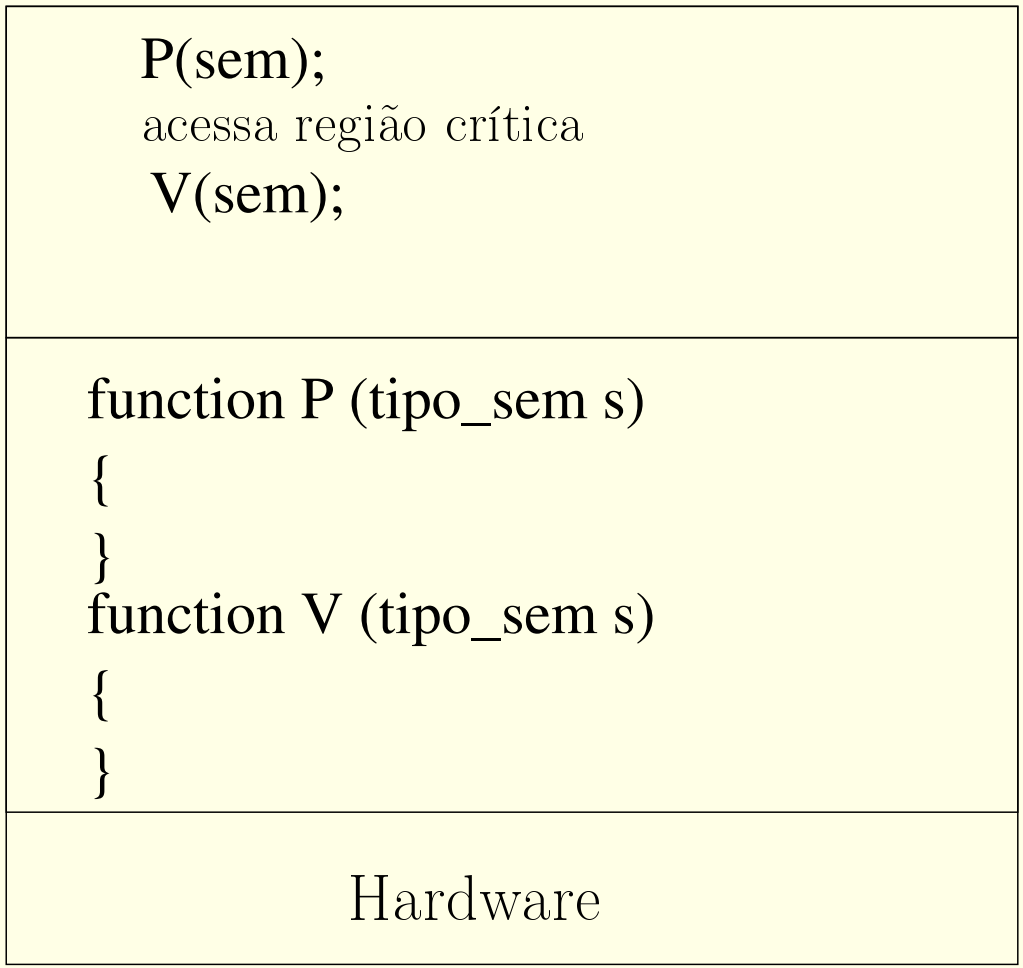
Até agora, vimos que o método de desabilitar interrupções, o método de estrita alternância e o uso da instrução TSL resolvem o problema do acesso a região crítica. Infelizmente, todos eles incorrem em problemas.

Semáforos

Semáforos são primitivas que podem ser escritas utilizando alguns dos métodos já vistos (tipicamente desabilitação de interrupções e instrução TSL). Assim, utilizando estes métodos são escritas duas funções `P()` e `V()`.

UTFPR

Comunicação entre processos - Semáforos



Programas de usuário que usam semáforos

Biblioteca que cria os semáforos usando algum método como desabilitar interrupções

Comunicação entre processos - Semáforos

A função `P(semáforo)` ou `Down(semáforo)` decrementa uma variável do tipo semáforo. Quando tenta-se decrementar uma variável do tipo semáforo cujo valor é 0, o processo que executou a tentativa de decremento é suspenso, sendo inserido numa lista de processos bloqueados.
A função `V(semáforo)` ou `Up(semáforo)` incrementa o valor de uma variável do tipo semáforo. Quando tentamos incrementar uma variável semáforo que estava com valor negativo, o processo que estava na lista de processos bloqueados passa para a lista de processos prontos e poderá ser executado novamente.

UTFPR

Comunicação entre processos - Semáforos

Valor dos semáforos - EXEMPLO

Uma operação de inicialização de semáforos sem=4 faz com que este seja o valor do semáforo. Ou seja, é possível que um processo execute `P(sem)` até quatro vezes antes de que este processo fique bloqueado. Se outro processo no sistema executar `V(sem)` fará com que o semáforo seja incrementado e liberará o processo que esteja dormindo.

Comunicação entre processos

```
1 void P (semaforo s)
2 {
3   if (s > 0)
4   s = s - 1;
5 else
6   sleep();
7 }
```

```
1 void V (semaforo s)
2 {
3   if (ExisteProcessoEsperando(s))
4     AtivaProcessoEsperando(s);
5   else
6     s = s + 1;
7 }
```

Comunicação entre processos - Semáforos

Semáforos binários ou mutex

Semáforos cujo valor foi inicializado para 1, ou seja permite que seja executado uma vez uma operação `P(sem)`, se uma segunda chamada for realizada, o processo será bloqueado.

Comunicação entre processos - Deadlocks

Deadlock

Deadlock é a situação onde temos processos concorrentes num computador e estes processos compartilham algum recurso. O controle de acesso a este recurso é feito por um semáforo e por algum erro, ambos os processos ficam bloqueados esperando a oportunidade de acessar o recurso.

Comunicação entre processos - Deadlock

```
1  semaforo sem_1=1;
2  semaforo sem_2=1;
3
4  void A (void)
5  {
6      while (1)
7      {
8          P(&sem_1);
9          P(&sem_2);
10         acessa_regiao_critica();
11         V(&sem_2);
12         V(&sem_1);
13     }
14 }
```

```
1
2
3  void B (void)
4  {
5      while (1)
6      {
7          P(&sem_2);
8          P(&sem_1);
9          acessa_regiao_critica();
10         V(&sem_1);
11         V(&sem_2);
12     }
13 }
```

Comunicação entre processos - Problema do produtor-consumidor

```
#define N 10
semaforo mutex=1;
semaforo vazio=N;
semaforo cheio=0;

void produtor (void)
{
    int item;
    while (1)
    {
        produz_algum_item();
        P(&vazio);
        P(&mutex);
        insere_item_produzido();
        V(&mutex);
        V(&cheio);
    }
}
```

```
1
2
3  void consumidor (void)
4  {
5      int item;
6      while (1)
7      {
8          P(&cheio);
9          P(&mutex);
10         remove_item();
11         V(&mutex);
12         V(&vazio);
13         consome_item(item);
14     }
15 }
```

Deadlocks

Imagine que um programador escreveu um código para o produtor, utilizando semáforos, mas acabou invertendo a ordem das operações de Down(). O que ocorre quando o produtor já produziu N elementos ?

```
1  void produtor (void)
2  {
3     int item;
4     while (1)
5     {
6        produz_algum_item();
7        P(&mutex);       // faz o down sobre mutex
8        P(&vazio);
9        insere_item_produzido();
10       V(&mutex);
11       V(&cheio);
12    }
13 }
```

Prática: Programação concorrente em C e semáforos

Pthreads

Como uma prática, veremos como pode-se trabalhar com programação concorrente na linguagem C. Como a linguagem C não possui natualmente recursos para permitir a execução "paralela" de funções, utilizaremos uma biblioteca chamada pthreads.

A biblioteca pthreads está disponível em diversos sistemas operacionais. Na aula prática será demonstrado o uso das pthreads em programas C no ambiente Windows e será mostrado também como utilizá-la no Linux.

Problema do produtor-consumidor

Contadores de eventos

O problema do produtor consumidor foi resolvido usando uma exclusão mútua (usando semáforos). O mesmo problema pode ser resolvido utilizando **contadores de eventos**. Um contador de evento é uma variável especial (compartilhada por todos os processos ou threads) que suporta algumas operações atômicas.

- READ(E) - Lê o valor da variável evento;
- ADVANCE(E) - Incrementa o valor da variável evento em 1 unidade. É uma função atômica;
- AWAIT(E,V) - Fica suspenso, esperando até que o valor E seja maior ou igual a V.

Problema do produtor-consumidor - contadores de eventos

```
1  #define N 10
2
3  event_counter in=0, out=0;
4
5
6
7  void produtor (void)
8  {
9      int item, sequence=0;
10     while (1)
11     {
12         produz_algum_item();
13         sequence++;
14         AWAIT(out,sequence-N);
15         insere_item_produzido();
16         ADVANCE(&in);
17     }
18 }
```

```
1
2
3  void consumidor (void)
4  {
5      int item, sequence=0;
6      while (1)
7      {
8          sequence++;
9          AWAIT(in,sequence);
10         remove_item();
11         ADVANCE(&out);
12         consome_item(item);
13     }
14 }
```

Comunicação entre processos - Monitores

Monitores

O semáforo é uma ferramenta bastante poderosa para controlar o acesso exclusivo a região crítica por processos concorrentes. Mas ele permite que por algum descuido tenhamos um **deadlock**. Para facilitar a escrita de programas concorrentes foi criado o conceito de **monitor**.

Comunicação entre processos - monitores - exemplo simples

```
1  monitor exemplo   // inicia o monitor
2      int count=0;
3
4     int   enter (x)  {
5        if (count < N) {
6           insere_item_produzido(x);
7           count++;
8           return true;
9        }
10       return false;
11
12    }
13    int remove(x) {
14       if (count >0)
15       {
16          remove_item(&x);
17          count--;
18          return true;
19       }
20       return false;
21    }
22 } // termina o monitor
```

```
1     void produz () {
2        int retorno;
3        while (true) {
4           produz_algum_item(x);
5           retorno=false;
6           while (retorno==false)
                 retorno=exemplo.enter(x)
                 ;
7        }
8     }
9     void consome ()  {
10       int retorno;
11       while (true)  {
12          retorno=false;
13          while (retorno==false) retorno
                 =exemplo.remove(&x);
14          remove_item(x);
15       }
16    }
```

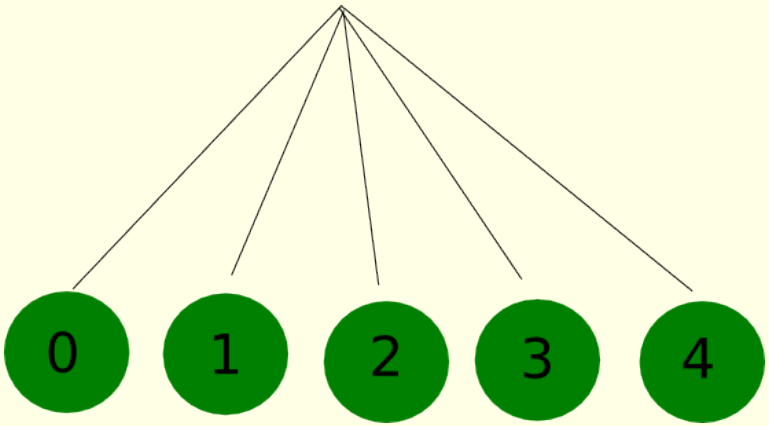
Comunicação entre processos - monitores - signal and wait

```
1  monitor exemplo   // inicia o monitor
2     int count=0;
3     condition full, empty;
4     void  enter (x)  {
5        if (count == N) wait(full);
6           insere_item_produzido(x);
7           count++;
8        if (count==1) signal(empty);
9     }
10    void remove(x) {
11       if (count ==0) wait(empty);
12          remove_item(&x);
13          count--;
14
15       if (count==N-1) signal(full);
16    }
17 } // termina o monitor
```

```
1     void produz () {
2        while (true) {
3           produz_algum_item(x);
4           exemplo.enter(x);
5        }
6     }
7     void consome ()  {
8        while (true)  {
9           exemplo.remove(&x);
10          remove_item(x);
11       }
12    }
```

Problema dos filósofos

Um problema clássico denominado problema dos filósofos jantando (*Dining philosophers problem*) apresenta o problema de compartilhamento de recursos limitados para processos. O objetivo é permitir que os processos consigam realizar suas tarefas sem a ocorrência de deadlocks/starvation.
Para deixar o problema mais divertido este é apresentado de uma uma forma curiosa: uma mesa de jantar com 5 filósofos (que fazem o papel de processos) e para conseguir comer um filósofo precisa de 2 garfos (recurso compartilhado), pela quantidade de garfos existentes na mesa(5), fica claro que não é possível que todos comam ao mesmo tempo.




```
1  #define N 5
2
3  for (int x=0;x<N;x++)
4     create_thread(philosopher,x);
```

```
1  void philosopher (int i)
2  {
3      while (1)
4      {
5          think();
6          take_fork(i);
7          take_fork(  (i+1) % N  );
8          eat(); // usa os recursos
9          put_fork(i);
10         put_fork(  (i+1)%N );
11     }
12 }
13 // Solucao errada
```

```
#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 */
    }
}
```

```
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]);
        }
}
```

Leitores e Escritores

Outro problema clássico é o dos leitores e escritores e serve para modelar o problema de existirem processos que precisam ler uma informação e outros processos (possivelmente apenas um) que precisa escrever uma informação.

Uma forma trivial de resolver o problema é permitir que exista apenas um processo leitor e apenas um processo escritor.

```
1  semaphore mutex = 1;
2  void Leitor (void) {
3     while(true) {
4        down(mutex);
5        le_dados();
6        up(mutex);
7     }
8  }
9  void Escritor (void) {
10    while(true) {
11       down(mutex);
12       escreve_dados();
13       up(mutex);
14    }
15 }
```

Não é possível que mais de um leitor leia a informação simultaneamente.

Solução final do problema: Vários leitores podem ser simultaneamente a informação.

```
1  semaphore mutex = 1, db    = 1;
2  int rc          = 0;
3  void Leitor (void) {
4     while(true) {
5        down(mutex);
6        rc++;
7        if (rc==1) down(db);
8        up(mutex);
9        le_dados();
10       down(mutex);
11       rc--;
12       if (rc==0) up(db);
13       up(mutex);
14       usa_dos_dados_lidos();
15    }
16 }
17 void Escritor (void) {
18    while(true) {
19       down(db);
20       escreve_dados();
21       up(db);
22    }
23 }
```