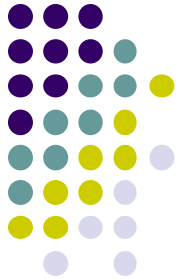
Comunicação Interprocessos

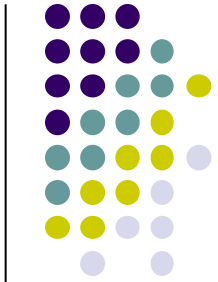
Troca de Mensagens



- Assume que existe alguém pronto para responder imediatamente
- Comunicação Bloqueante: Quem solicita aguardando a resposta
  - Ex: Web-Services, RPC, Sockets TCP/IP, MPI
- RPC – Sun Microsystems
  - Um processo que invoca procedimentos do outro remotamente
- Sockets TCP/IP
  - comunicação síncrona full-duplex
- Web-Services
  - Padrão aberto multiplataforma para troca de mensagens
  - Dados encapsulados em pacotes XML/JSON transparentemente
- MPI – Message Passing Interface
  - API para C e Fortran para comunicação explícita entre processos
  - Uso em computação científica

Comunicação Interprocessos

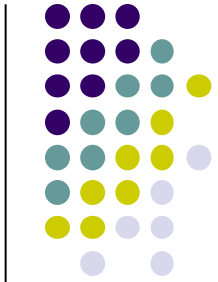
Entrada e Saída



- Uso de pipes ou named pipes
  - FIFO: First In First Out
    - O primeiro dado escrito pelo processo produtor é o primeiro a ser lido pelo consumidor
  - Named pipe
    - Um arquivo fictício que pode ser aberto por 2 processos distintos
    - # mkfifo <nomearq>
    - Um processo lê (fread) e outro escreve (fwrite) no arquivo
    - O tamanho do arquivo nunca ultrapassa 0 bytes
  - Pipe
    - O arquivo não é criado físicamente, apenas em memória enquanto os processos estão trocando informações
    - # cat listaTelefonica.txt | sort –u
      - Envia o conteúdo de listaTelefonica.txt para o comando sort ordenar e exibir somente as entradas únicas.

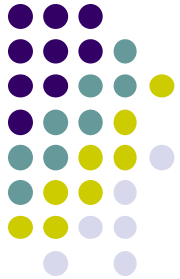
Comunicação Interprocessos

Recurso Compartilhado



- Leitura e escrita concorrente.
- Possibilidade de colisão de acesso aos dados
  - Vários processos atuando sobre a mesma área de memória
  - TERMO USADO EM S.O.s: “CONDIÇÃO DE DISPUTA” ou “RACE CONDITION”
- Exemplos de mecanismos existentes
  - Malloc especial para reservar memória para 2 ou mais processos
  - Variáveis compartilhadas entre vários threads

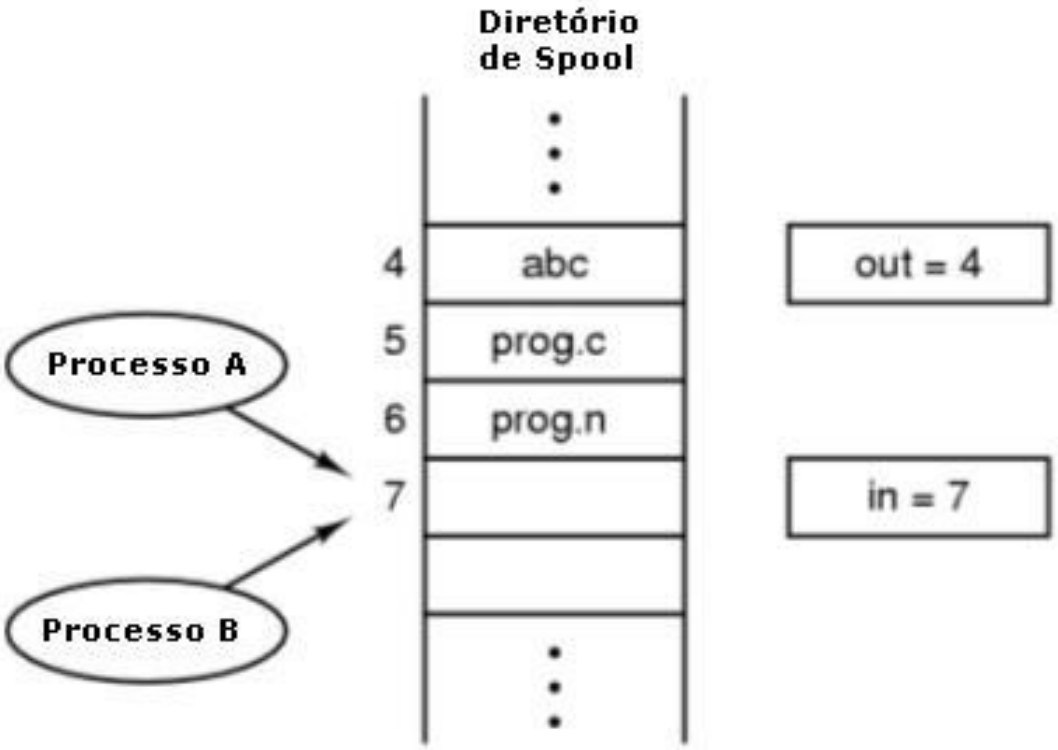
Comunicação Interprocessos
Condições de Disputa



- IPC (Inter Process Communication)
  - Conceitos se aplicam igualmente à threads
  - Pode ser difícil se resolver problemas deste tipo
- Como ocorre?
- Como garantir consistência espacial
  - Um processo não deve invadir o outro
  - Ex: Evitar que dois processos usem o mesmo lugar na fila para armazenar um trabalho de impressão
- Como garantir consistência temporal
  - Seqüenciamento
  - Ex: Um *daemon* de impressão só pode imprimir depois que o arquivo foi inteiramente colocado na fila de impressão

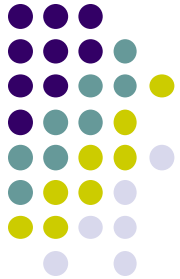
Comunicação Interprocessos

Condições de Disputa (2)



Dois processos querendo acessar memória compartilhada ao mesmo tempo

Comunicação Interprocessos
Condições de Disputa (3)



- Como evitar?
  - Evitar que a memória compartilhada seja acessada por mais de um processo por vez
  - Isso se chama “Exclusão Mútua” (mutual exclusion)
  - Criação de regiões ou seções críticas quando houver acesso à memórias compartilhadas

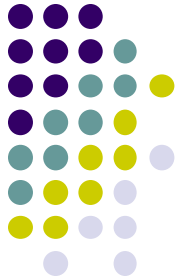
Objetivo

Evitar que 2 ou mais processos estejam em suas regiões críticas ao mesmo tempo

Exclusão Mútua

Comunicação Interprocessos

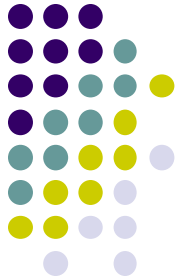
Regiões Críticas



- São trechos presentes no código-fonte de um programa onde ocorre o acesso a um recurso compartilhado
  - Existe possibilidade de acesso concorrente ao recurso compartilhado
  - É necessário existir regiões críticas quando um processo A estiver interagindo com um processo B através de um recurso compartilhado
    - Ex: Variável compartilhada

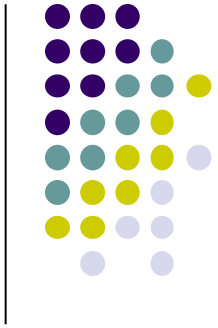
Comunicação Interprocessos

Regiões Críticas



- As regiões críticas de ambos os programas não podem executar ao mesmo tempo
  - Se o programa A estiver processando sua propria região crítica, o programa B não pode executar a sua região crítica também, e vice versa.
  - Isso se chama exclusão mútua
- As regiões críticas devem ser protegidas através de ferramentas disponibilizadas pelas linguagens de programação, em conjunto com o Sistema Operacional para prover mecanismos de exclusão mútua

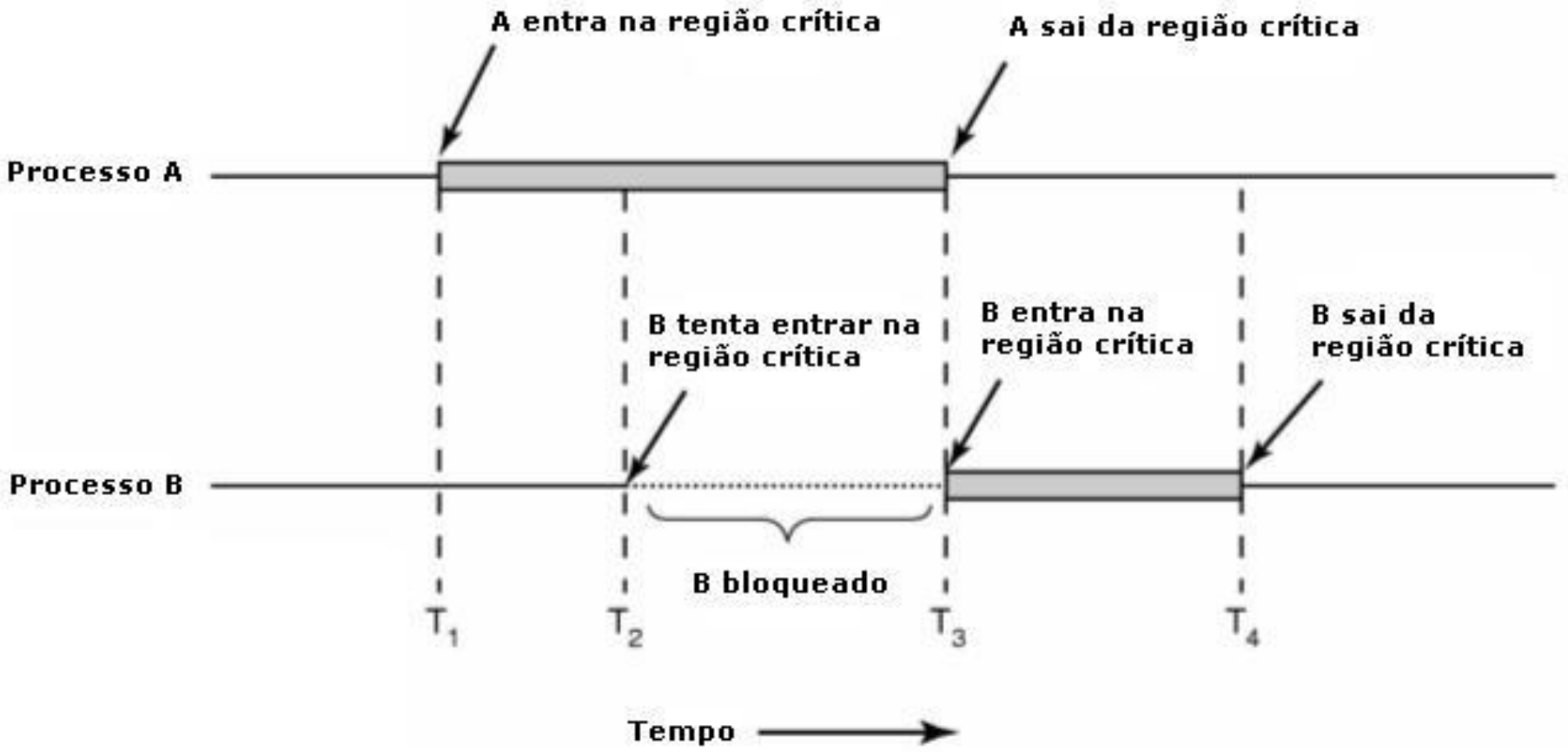
Regiões Críticas (1)



Quatro condições para uma boa solução

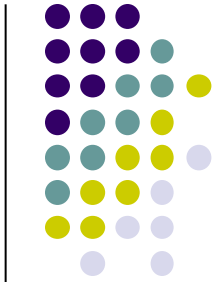
1. Nunca pode ocorrer de termos dois processos simultaneamente nas suas regiões críticas
2. Nada pode ser afirmado sobre velocidade ou número de CPUs
    1. A exclusão mútua deve funcionar sem que isso a prejudique
3. Nenhum processo fora de sua região crítica pode bloquear outros processos
4. Nenhum processo pode esperar para sempre para entrar na sua região crítica

Regiões Críticas (2)



Exclusão mútua usando regiões críticas

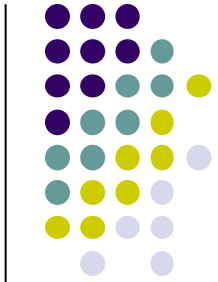
Mutexes
(mutex_lock e mutex_unlock)



- Proteção de regiões críticas
- Muito utilizado
- Simples

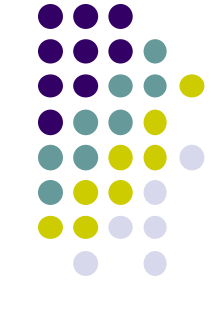
"O *mutex* funciona como uma trava parecida com as encontradas em armários públicos em aeroportos ou alguns bancos. Se a porta estiver aberta, é só usar (e trancar). Se estiver fechada você deve esperar a sua vez" (GUBITOSO, 2003)."

Mutexes (2)
(mutex_lock e mutex_unlock)



- Dois estados possíveis de um mutex:
  - Impedido: Recurso já está em uso
  - Desimpedido: Recurso não está sendo usado por ninguém

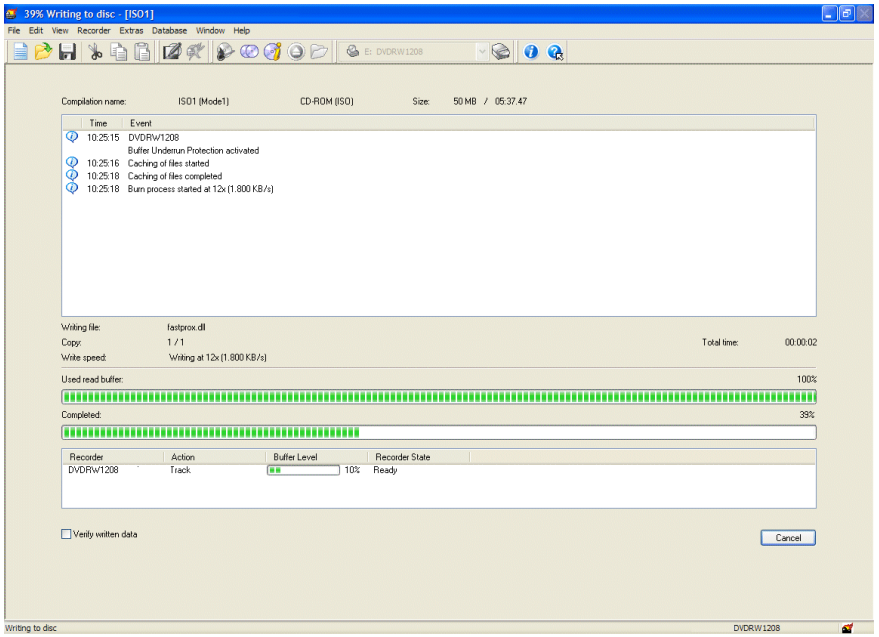
Mutexes (3)
(mutex_lock e mutex_unlock)



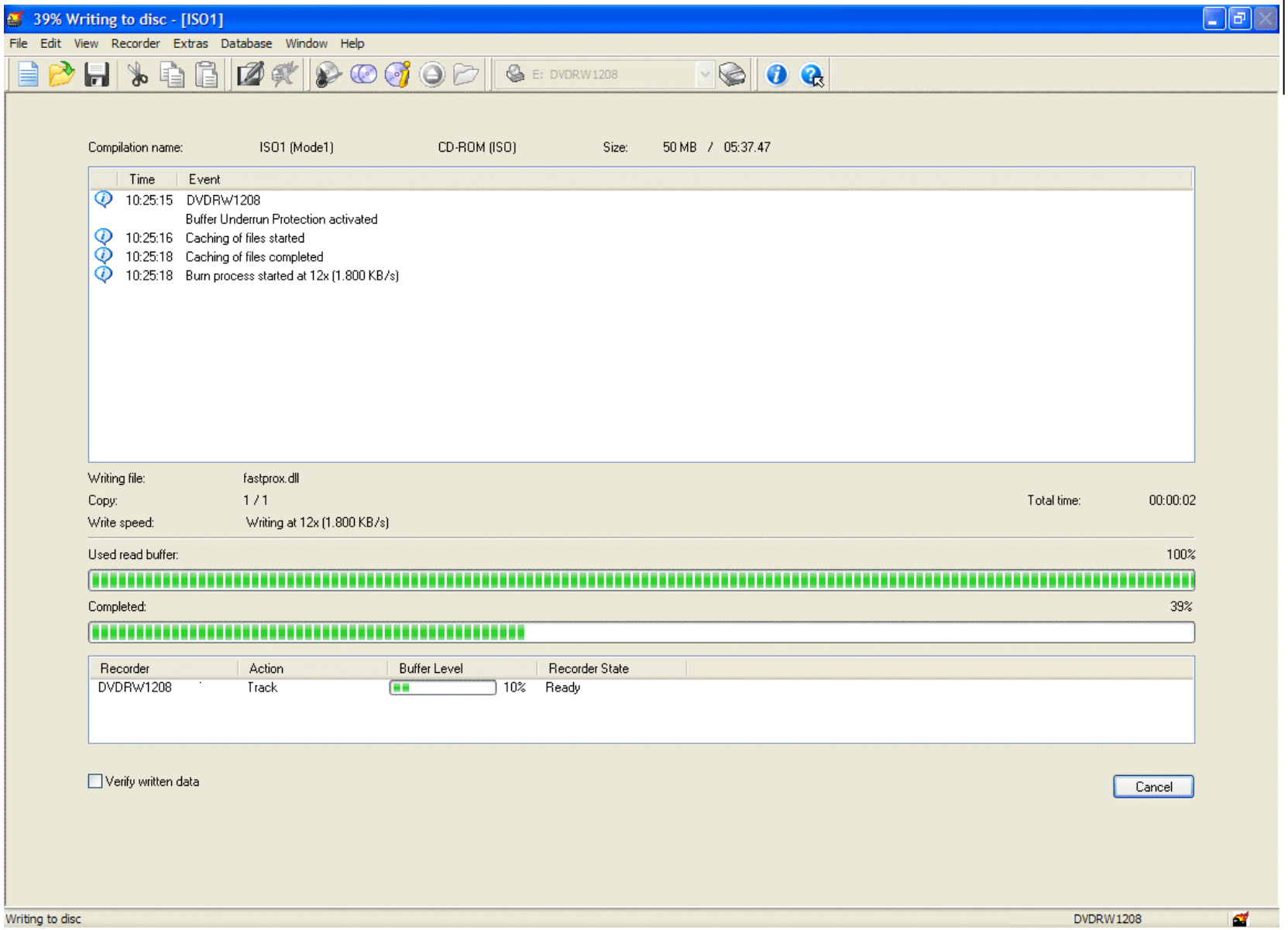
```
void *SomaElementos( void *id ) {
   int i, sp = 0, p = (int *)id;
   for( i = p; i < TAM_VETOR; i = i + NUM_THREADS ) {
       sp += elem[i];
   }
   pthread_mutex_lock( &mut );
   somatotal += sp;              // variável compartilhada
   pthread_mutex_unlock( &mut );
   pthread_exit(NULL);
}
```

Produtor x Consumidor

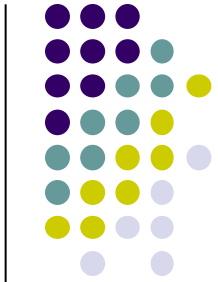
- Outro tipo de problema: Sincronização interprocessos
- Exemplo: Gravação de CDs ou DVDs
  - 2 threads
    - Produtor: Alimenta o buffer em memória que não pode ficar vazio
    - Consumidor: Usa dados do buffer para gravar dados na mídia



Produtor x Consumidor

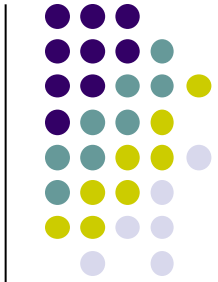


Produtor x Consumidor (3)



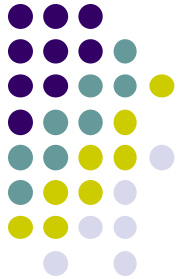
- Problema clássico de sincronização:
  - Produtor precisa ser bloqueado se o buffer estiver cheio
  - Consumidor precisa ser bloqueado se o buffer estiver vazio
  - Produtor acorda consumidor após colocar informações no buffer
  - Consumidor acorda produtor após consumir dados do buffer
    - *Obs: Uma solução aceitável não deve depender da velocidade de execução dos processos*

Sleep e Wakeup



- Primitivas de sincronização simples
- É uma técnica problemática
- Não funciona em um certo cenário o que inviabiliza o seu uso

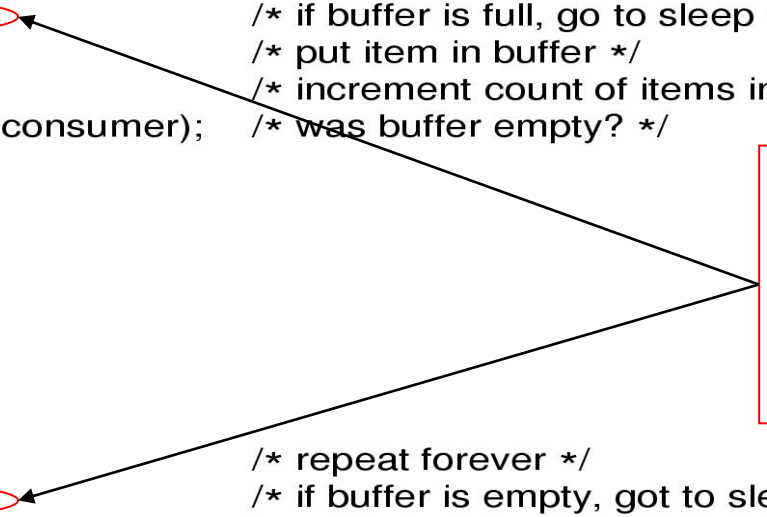
Sleep e Wakeup (2)



```
#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 */
     }
}
```

Sleep e Wakeup (2)

```
#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 */
     }
}
```



Problema: quantum do processo termina antes de invocar sleep.

`Produtor está executando e o consumidor também.
O buffer está cheio e a execução do produtor está nesse ponto

Sleep e Wakeup (2)

ele acabou de produzir um item

ele verifica que o buffer está cheio

antes de chamar sleep(), acaba o quantum do produtor

Produtor volta a ser executado e SLEEP!!!
Nada mais será produzido pois o p produtor não receberá outro wakeup

```
#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? */
     }
}
```

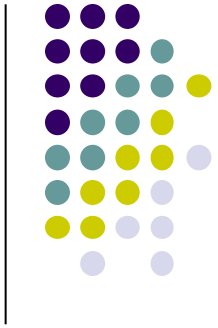
consumidor começa a executar
retira um item do buffer
decrementa o contador (ou seja o buffer agora tem espaço)
ele chama wakeup do produtor (pois já tem espaço livre!!)

O PROBLEMA É QUE O PRODUTOR AINDA NÃO DORMIU!!
O WAKEUP É PERDIDO

```
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 */
     }
}
```

Problema: quantum do processo termina antes de invocar sleep.

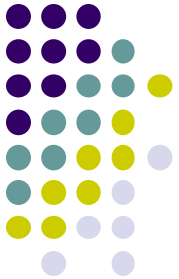
Sleep e Wakeup



- Primitivas de sincronização simples
- É problemática
  - Não funciona no seguinte cenário:
    - Buffer está cheio
    - Processo produtor executa `if (count == N)` e antes de invocar `sleep()` ele é preemptado e vai para o estado pronto
    - Processo consumidor executa, e executa `if (count == N - 1) wakeup(produtor)`
    - Produtor ainda não estava 'dormindo'. O sinal `wakeup` é perdido
    - Consumidor consome todo buffer. Produtor volta executando `sleep()`
    - Ambos entram em deadlock
    - Situação também chamada de starvation (2 processos 'passando fome')
  - Gerenciador de processos pode preemptar processo em qualquer lugar
  - A verificação do estado e invocação da primitiva sleep não é uma atividade atômica.

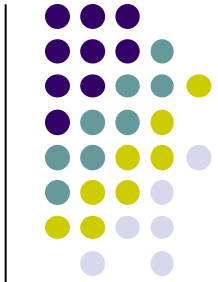
Semáforos (Up e Down)

Dijkstra (1965)



- Alternativa ao sleep() e wakeup()
- Uso das primitivas P (ou down) e V (ou up)
- Funciona mas é necessário ter suporte do SO.
  - Primitivas P e V devem ser atômicas
    - Atomicidade ➔ Não podem ser preemptadas pelo SO no meio de sua execução
    - O S.O. garante consistência temporal e espacial no uso da variável semáforo
- São necessários 2 semáforos para resolver problema produtor/consumidor
- O mutex é um semáforo simplificado!

Semáforos (2)



- Duas operações básicas:
  - P - DOWN

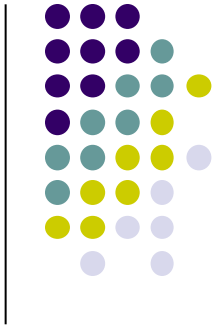
```
valid: se semáforo > 0 entao
           Decrementa semáforo
            Prossegue
         senão
            vai para estado bloqueado
            pula para vai para valid
       fim-se
```

Observe que usamos uma construção que não pertence à programação estruturada em negrito

  - V - UP

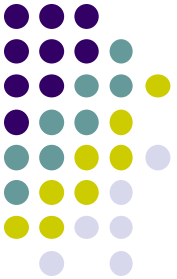
```
       se semáforo = 0 E houver procs. esperando semaforo
           desbloqueia algum dos processos
       senão
           incrementa o semáforo
       fim-se
```

Semáforos (3)



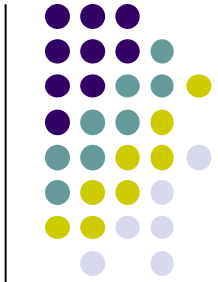
```
semaforo cheio = 0;
semaforo vazio = N;
void consumidor() {
    while (houverDados) {
        p(cheio) // down(cheio)
        // consome dados
        v(vazio) // up(vazio)
    }
}
void produtor() {
    while (houverDados) {
        p(vazio) // down(vazio)
        // produzir dados
        v(cheio) // up(cheio)
    }
}
void main() {
    // cria buffer onde os dados serão armazenados
    int pid = fork();
    if (pid == 0) {
        consumidor();
    } else {
        produtor();
    }
}
```

Semáforos usando PThreads



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

#define TAMANHOBUFFER 5
#define TAMANHOMIDIA 20

// inicializacao dos semaforos
pthread_mutex_t  mutexBuffer; // proteger buffer
sem_t cheio, livre;

// inicializacao do buffer
int buffer[TAMANHOBUFFER];
int quantBytesEscritos = 0;
int quantBytesLidos = 0;
```

Semáforos usando PThreads (2)



```
void *produtor( void *id ) {

    while( quantBytesEscritos < TAMANHOMIDIA ) {
        // se o buffer estiver cheio
        // aguarda um sinal do consumidor
        sem_wait(&livre);

        // Pegar a posicao do buffer que sera modificada
        int posicao = quantBytesEscritos % TAMANHOBUFFER;

        // Modificar o buffer
        pthread_mutex_lock(&mutexBuffer);
        buffer[posicao] = (int) rand(324);
        printf("Info colocada no buffer na posicao %d: %d\n", posicao, buffer[posicao]);
        pthread_mutex_unlock(&mutexBuffer);

        quantBytesEscritos++;

        sem_post(&cheio);
    }
    pthread_exit(NULL);
}
```

Semáforos usando PThreads (3)

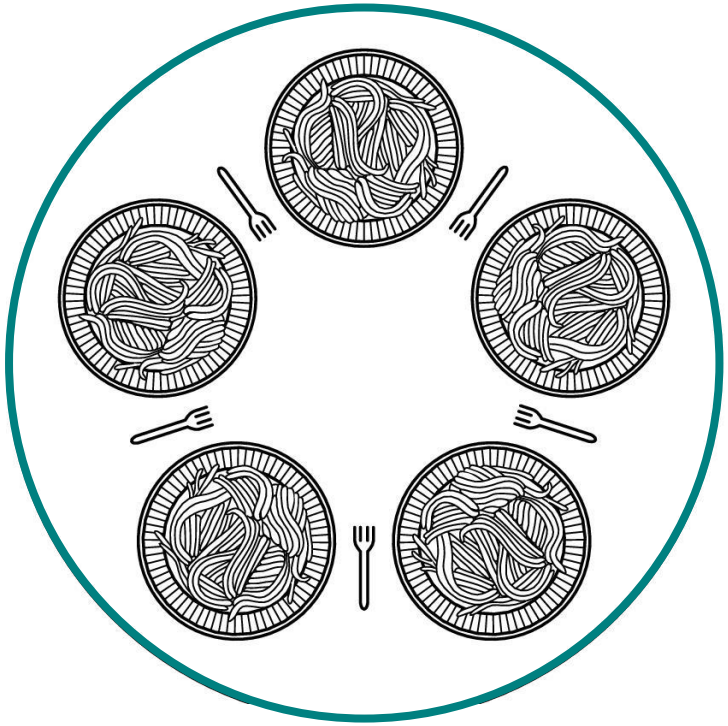
```
void *consumidor(void *id) {

    while (quantBytesLidos < TAMANHOMIDIA) {
        sem_wait(&cheio);

        // Pegar a posicao do buffer que sera lida
        int posicao = quantBytesLidos % TAMANHOBUFFER;

        // Retirar dados do buffer
        pthread_mutex_lock(&mutexBuffer);
        printf("Info retirada do buffer na posicao %d: %d\n", posicao,
   buffer[posicao]);
        pthread_mutex_unlock(&mutexBuffer);

        sleep(1000);     // va dormir por 2 segundos

        quantBytesLidos++;
        sem_post(&livre);
    }
    pthread_exit(NULL);
}
```

Semáforos usando PThreads (4)

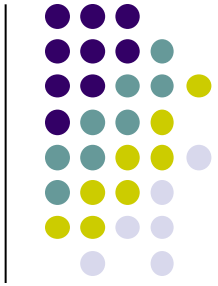
```
int main( int argc, char *argv[] ) {
    pthread_t tConsumidor, tProdutor;

    printf( "Inicializando Semaforos e mutex\n");
    sem_init(&cheio, 0, 0);
    sem_init(&livre, 0, TAMANHOBUFFER);
    pthread_mutex_init( &mutexBuffer, NULL );

    printf( "Criando thread produtor\n");
    int rc = pthread_create(&tProdutor, NULL, produtor, NULL);
    printf( "Criando thread consumidor\n");
    rc = pthread_create(&tConsumidor, NULL, consumidor, NULL);

    // aguarda todos os threads terminarem
    pthread_join(tProdutor, NULL);
    pthread_join(tConsumidor, NULL);

    printf("Processamento terminado: %d - %d\n", quantBytesEscritos, quantBytesLidos);

    pthread_mutex_destroy( &mutexBuffer );
    sem_destroy( &cheio );
    sem_destroy( &livre );
    getchar();
    pthread_exit( NULL );
}
```

Jantar dos Filósofos

- Cada filósofo possui um prato de espaguete
- Para comer o espaguete o filósofo precisa de dois garfos
- Existe um garfo entre cada par de pratos
- Um filósofo come ou medita
  - Quando medita não interage com seus colegas
  - Quando está com fome ele tenta pegar dois garfos um de cada vez. Ele não pode pegar um garfo que já esteja com outro filósofo
- Os garfos são os recursos compartilhados



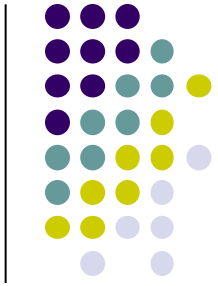
Jantar dos Filósofos



```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

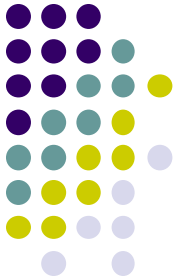
- Se todos pegam o garfo da esquerda ao mesmo tempo ocorrerá *deadlock ou impasse*.

Jantar dos Filósofos



```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N−1)%N          /* number of i's left neighbor */
#define RIGHT            (i+1)%N            /* number of i's right neighbor */
#define THINKING         0                  /* philosopher is thinking */
#define HUNGRY           1                  /* philosopher is trying to get forks */
#define EATING           2                  /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N−1 */
{
      while (TRUE) {                        /* repeat forever */
            think( );                       /* philosopher is thinking */
            take_forks(i);                  /* acquire two forks or block */
            eat( );                         /* yum-yum, spaghetti */
            put_forks(i);                   /* put both forks back on table */
      }
}
```

Jantar dos Filósofos



```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = THINKING;            /* philosopher has finished eating */
        test(LEFT);                     /* see if left neighbor can now eat */
        test(RIGHT);                    /* see if right neighbor can now eat */
        up(&mutex);                     /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```