Sistemas Operacionais: IPC

Prof. Dr. Rafael Lopes Gomes
Universidade Estadual do Ceará (UECE)

Agenda

- Processos
- Threads
- **Comunicação entre processos**
- Escalonamento de Processos

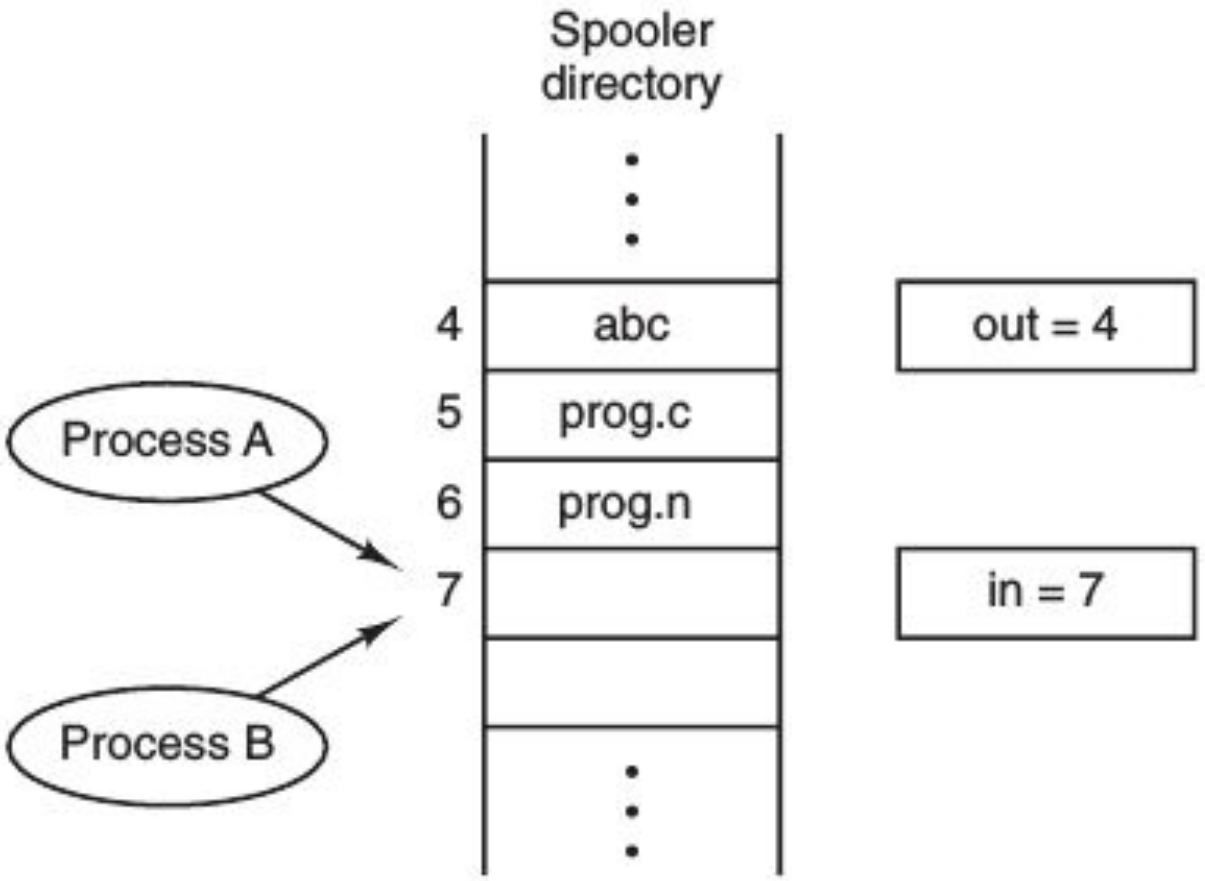
Inter Process Communication (IPC)

- Comunicar processos sem interrupções (parada explícita de execução na CPU)
- Aspectos principais:
  - Como um processo pode passar informações para outro?
  - Como certificar-se que dois ou mais processos não se atrapalhem ?
  - Como garantir um sequenciamento adequado quando as operações são dependentes ?

Condições de Corrida

- Processos que estão trabalhando em conjunto podem compartilhar memória (principal ou disco) para ler e escrever.
- **Condições de corrida**: Situações em que dois ou mais processos (ou threads) estão lendo ou escrevendo alguns dados compartilhados, dependem de quem e quando executa.

Condições de Corrida



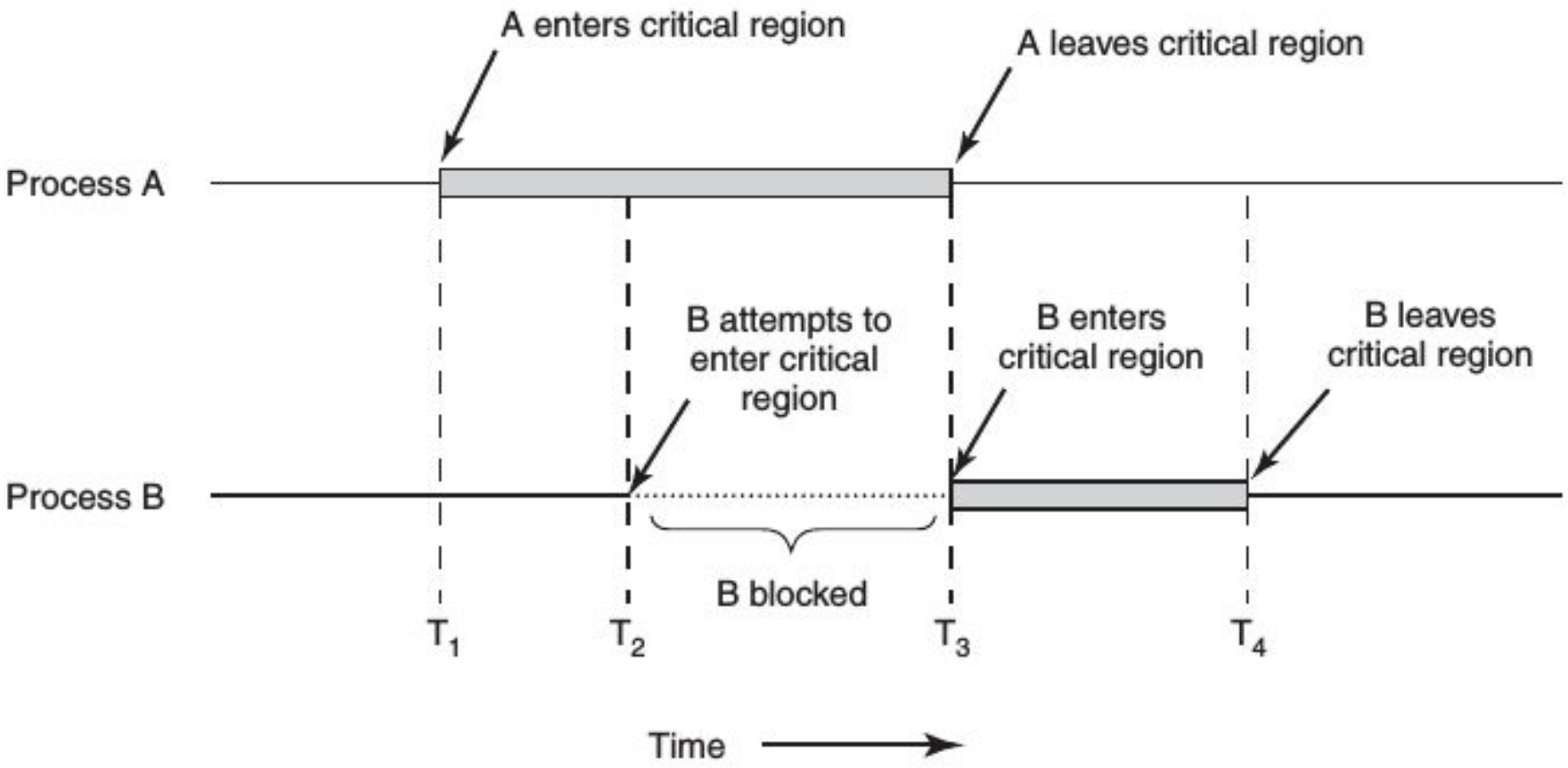
Regiões Críticas

- Para evitar problemas em condições de corrida, é preciso proibir que mais do que um processo leia e escreva ao mesmo tempo.
- **Exclusão mútua**: certificar que se um processo está usando a memória compartilhada, os outros estão impedidos de realizar alguma operação.
- **Região Crítica (ou Seção Crítica)**: é a parte do programa onde a memória compartilhada é acessada.

Regiões Críticas

- Quatro condições para evitar condições de corrida:
  - Dois processos jamais podem estar simultaneamente dentro de suas regiões críticas;
  - Nenhuma suposição pode ser feita a respeito de velocidades ou do número de CPUs;
  - Nenhum processo executando fora de sua região crítica pode bloquear qualquer outro processo;
  - Nenhum processo deve ser obrigado a esperar infinitamente para entrar em sua região crítica.

Regiões Críticas



Exclusão Mútua (EM)

- No geral, há duas estratégias para realizar exclusão mútua:
  - Espera Ocupada
    - Processo checa constantemente se já pode acessar a região.
  - Dormir e acordar
    - Processo é bloqueado e acorda quando o outro processo sai da região crítica.

Exclusão Mútua: Estratégias

- Desabilitando Interrupções
  - Único processador: processo desabilita todas as interrupções logo após entrar na região crítica e reabilita antes de sair;
  - Desvantagem:
    - Não é prudente dar aos processos esse poder;
    - Não funciona em sistemas multiprocessados: desabilitar uma CPU não influencia as demais.

Exclusão Mútua: Estratégias

- Variáveis do Tipo Trava
  - Quando um processo quer entrar em sua região crítica, ele primeiro testa e trava.
  - Se a trava é 0 (zero), o processo a configura para 1 e entra na região crítica;
  - Se a trava já é 1, o processo apenas espera até que ela se torne 0.

Exclusão Mútua: Estratégias

- Variáveis do Tipo Trava: problema
  - Suponha que um processo lê a trava e vê que ela é 0.
  - Antes que ele possa configurar a trava para 1, outro processo está escalonado, executa e configura a trava para 1.
  - Quando o primeiro processo executa de novo, ele também configurará a trava para 1, e dois estarão nas suas regiões críticas ao mesmo tempo.
  - **Quebra da condição 1.**
    - “Dois processos jamais podem estar simultaneamente dentro de suas regiões críticas”

Exclusão Mútua: Estratégias

- Alternância explícita
  - Tem-se uma variável (***turn***), inicialmente 0, serve para controlar de quem é a vez de entrar na região crítica.
  - Estratégia: **Espera Ocupada**
    - Testar continuamente o valor da variável até que o valor apareça.
  - Problema: processos com “velocidade” muito diferente.

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

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

Exclusão Mútua: Estratégias

- Solução de Peterson
  - Estratégia: antes de entrar na região crítica, cada processo chama uma função de entrada na região (*enter_region*) com seu próprio número de processo (0 ou 1).
  - A chamada faz o processo esperar até que seja seguro entrar.
  - Após realizar suas ações, o processo chama uma função de saída (*leave_region*) para permitir que outros processos entrem na região crítica.

Exclusão Mútua: Estratégias

```
#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: Dormir e Acordar

- Até agora, todas as soluções funcionam como espera ocupada
  - Testam constantemente os valores
  - Competem por CPU desnecessariamente
- Problemas
  - Desperdiça tempo de CPU
  - Comportamento inesperado (problema de inversão de prioridade)
- Estratégia: bloquear processos ao invés de desperdiçar tempo de CPU.

Exemplo: inesperada.c

```
volatile int s = 0; /* Variável compartilhada */

void* f_thread_0(void *v) {
	s = 0;
	printf("Thread 0, s = %d.\n", s);
	return NULL;
}
void* f_thread_1(void *v) {
	s = 1;
	printf("Thread 1, s = %d.\n", s);
	return NULL;
}
int main() {
	pthread_t thr0, thr1;
	pthread_create(&thr1, NULL, f_thread_1, NULL);
	pthread_create(&thr0, NULL, f_thread_0, NULL);
	pthread_join(thr0, NULL);   pthread_join(thr1, NULL);
	return 0;
}
```

Exemplo: tentativa_lock.c

```
volatile int s = 0; /* Variável compartilhada */
volatile int lock = 0;
void* f_thread_0(void *v) {
  while (lock != 0); /* Espera a "obtenção" do lock */
  lock = 1;
  s = 0;
  printf("Thread 0, s = %d.\n", s);
  lock = 0;  /* Libera o lock */
  return NULL;
}
void* f_thread_1(void *v) {
  while (lock != 0); /* Espera a "obtenção" do lock */
  lock = 1;
  s = 1;
  sleep(1);
  printf("Thread 1, s = %d.\n", s);
  lock = 0;  /* Libera o lock */
  return NULL;
}
```

Exemplo: alternancia.c

```
volatile int s = 0; /* Variável compartilhada */
volatile int vez = 0; /* Indica de qual thread é a vez de entrar na região crítica */

void* f_thread_0(void *v) {
  int i;
  for (i = 0; i < 5; i++) {
    while (vez != 0); /* Espera ser a vez desta thread */
    s = 0;
    printf("Thread 0, s = %d.\n", s);
    vez = 1; /* Passa a vez para a outra thread */
    sleep(1);
  }
  return NULL;
}
void* f_thread_1(void *v) {
  int i;
  for (i = 0; i < 5; i++) {
    while (vez != 1); /* Espera ser a vez desta thread */
  s = 1;
  printf("Thread 1, s = %d.\n", s);
  vez = 0; /* Passa a vez para a outra thread */
  }
  return NULL;
}
```

Exemplo: interesse2.c

```
volatile int s = 0; /* Variável compartilhada */
volatile int interess[2] = {0, 0};
void* f_thread_0(void *v) {
  int i;
  for (i = 0; i < 10; i++) {
     interesse[0] = 1; /* Marca que esta thread está interessada */
    while (interesse[1]) {
      interesse[0] = 0;
      sleep(1);
      interesse[0] = 1;
}
    s = 0;
    printf("Thread 0, s = %d.\n", s);
       interesse[0] = 0;  /* Marca que saiu da região crítica */
  }
  return NULL;
}
```

```
void* f_thread_1(void *v) {
  int i;
  for (i = 0; i < 10; i++) {
     interesse[1] = 1; /* Marca que esta thread está interessada */
    while (interesse[0]) {
      interesse[1] = 0;
      sleep(1);
      interesse[1] = 1;
    }
    s = 1;
    printf("Thread 1, s = %d.\n", s);
     interesse[1] = 0;  /* Marca que saiu da região crítica */
  }
  return NULL;
}
```

Exemplo: interesse_vez.c

```
volatile int s = 0; /* Variável compartilhada */
volatile int vez = 1;
volatile int interesse[2] = {0, 0};

void* f_thread_0(void *v) {
  int i;
  for (i = 0; i < 10; i++) {
    interesse[0] = 1; /* a thread está interessada */
    while (interesse[1] && vez != 0);
    s = 0;
    printf("Thread 0, s = %d.\n", s);
    vez = 1;          /* Passa a vez*/
    interesse[0] = 0;  /* Marca que saiu */
    sleep(1);
  }
  return NULL;
}
```

```
void* f_thread_1(void *v) {
  int i;
  for (i = 0; i < 10; i++) {
    interesse[1] = 1; /* a thread está interessada */
    while (interesse[0] && vez != 1);
    s = 1;
    printf("Thread 1, s = %d.\n", s);
    vez = 0;          /* Passa a vez*/
    interesse[1] = 0;  /* Marca que saiu */
    sleep(2);
  }
  return NULL;
}
```

Problema do Produtor-Consumidor

- Dois processos compartilham de um buffer de tamanho fixo.
  - Produtor: processo que insere informações
  - Consumidor: processo que retira informações
  - Uma variável (*count*) para indicar quantos itens existem no buffer.
  - Produtor incrementa *count*
  - Consumidor decrementa *count*
- Funcionamento:
  - Quando o produtor quer colocar um novo item no buffer, mas ele já está cheio.
  - Produtor é colocado pra dormir, e é despertado quando o consumidor tiver removido um ou mais itens.
  - Se o consumidor verificar que o buffer está vazio, ele dorme até o produtor colocar algo no buffer e despertá-lo.

Problema do Produtor-Consumidor

```
#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 */
     }
}
```

```
#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:
  - Caso o buffer esteja vazio e o consumidor leia *count* = 0, e simultaneamente a CPU para de executar o consumidor e executa o produtor.
  - O produtor insere um item no buffer, incrementa o count.
  - Como era 0, o produtor supõe que o consumidor está dormindo e chama *wakeup*.
  - Infelizmente, o consumidor ainda não está dormindo, então o sinal de despertar é perdido.
  - Assim, quando volta a execução, o consumidor testará o valor que leu antes (zero) e irá dormir infinitamente.
  - Em algum momento o produtor irá encher o buffer e dormirá infinitamente também

Semáforos

- Semáforo: variável inteira para contar o número de sinais de acordar salvos.
  - Zero: nenhum sinal de despertar salvo;
  - Positivo: sinais pendentes.
- Operações em semáforos:
  - Down: confere se o valor é positivo, usa um sinal de acordar (decrementa). Se o valor for zero, o processo dorme, sem concluir a operação down;
  - Up: se um ou mais processos estiverem dormindo no semáforo (incapaz de completar um down), um deles é autorizado a completar o down.

Semáforos

- Conferir o valor, modificá-lo e possivelmente dormir são feitos como uma única operação atômica.
- A operação de incrementar o semáforo e despertar um processo também é atômica.
- Ação atômica: grupo de operações relacionadas são todas realizadas sem interrupção.
  - Indivisível
  - É garantido que quando a operação do semáforo tenha começado, nenhum outro processo pode acessá-lo até a operação ser concluída ou bloqueada

```
#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 */
     }
}
```

Semáforos

- Se cada processo realiza um down um pouco antes de entrar em sua região crítica e um up logo após deixa-lá, a exclusão mútua é garantida.
- Semáforo mutex: garante que apenas um processo por vez esteja lendo ou escrevendo no buffer.
- Semáforos full e empty: asseguram que o produtor pare de executar quando o buffer estiver cheio, e que o consumidor pare de executar quando ele estiver vazio.

Exemplo: sem-ex.c

```
#define N 10
sem_t sem;
void* thread(void* v){
    int i = *(int*)v;
    sem_wait(&sem);
    printf("%d :: In\n",i);
    sleep(1);
    printf("%d :: Out\n",i);
    sem_post(&sem);
}
int main(){
  sem_init(&sem, 0, 1);  pthread_t thr[N];  int id[N], k;
  for (k = 0; k < N; k++) {
    id[k] = k;
    pthread_create(&thr[k], NULL, thread, &id[k]);
  }
  for (k = 0; k < N; k++)
    pthread_join(thr[k], NULL);

sem_destroy(&sem);
return 0;
}
```

Mutexes

- Quando não é preciso que um semáforo faça contagem, tem-se uma versão simplificada, o **mutex**
  - Bons para gerenciar exclusão mútua de algum recurso;
  - Variável compartilhada que pode estar travada ou destravada;
- Rotinas
  - Lock: requisita acesso
    - Destravado: chamada segue e a thread entra na região crítica;
    - Travado: thread é bloqueada, até que a thread na região crítica conclua a execução e chame um unlock.
  - Unlock: libera o acesso

Monitores

- Coleção de rotinas, variáveis e estruturas de dados que são reunidas em um tipo especial de módulo/pacote;
- Processos chamam essas rotinas, mas não tem acesso direto as estruturas de dados internos do monitor.
- Cabe ao compilador implementar a exclusão mútua nas entradas do monitor, mas uma maneira comum é usar mutex ou semáforo.
- Problema:
  - A linguagem precisa ter suporte
  - Projetados, assim como semáforos, para tratar exclusão mútua em máquina local (sem suporte a SD).

Troca de Mensagens

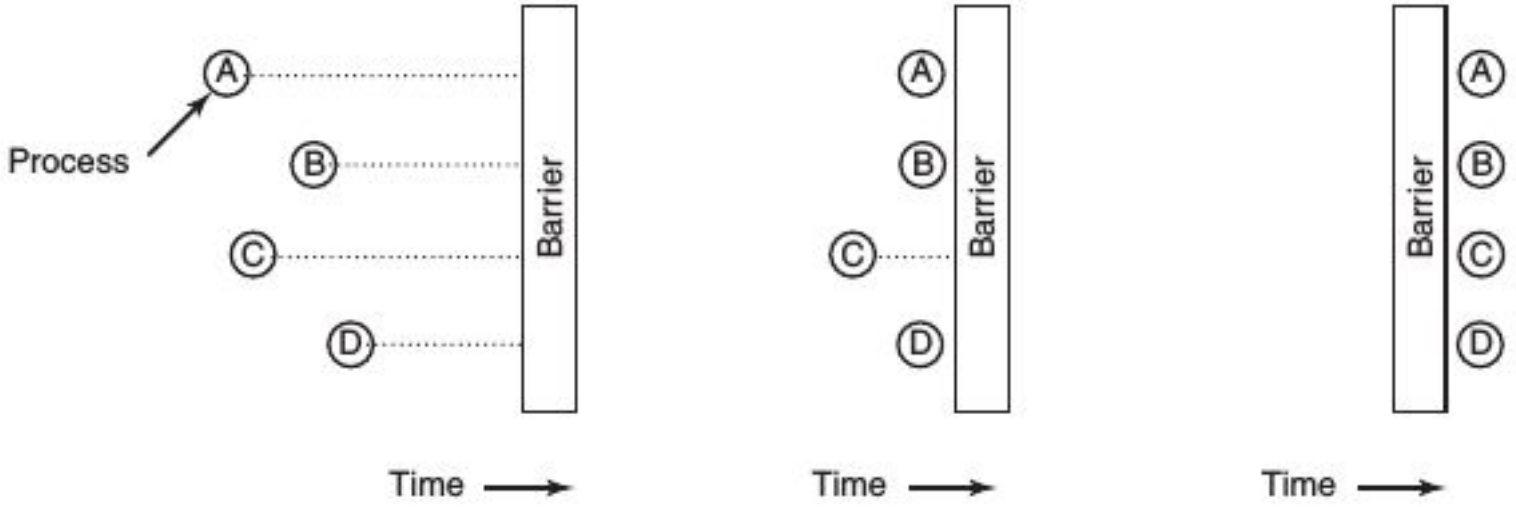
- Método de comunicação entre processos com duas primitivas: send e receive;
- Problemas:
  - Possível perda de mensagens
    - Confirmação ?
    - Duplicação ?
  - Como identificar processos ? Autenticação ?
  - Desempenho
    - Mais lento

Troca de Mensagens

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Barreiras

- Aplicação dividida em fases, onde todos os processos precisam passar de fase juntos;
- Inserir uma barreira no final de cada fase;
- Quando um processo atinge a barreira, ele é bloqueado até que todos alcancem a barreira.



Exemplo: cond_signal_n.c

```
/* Flags auxiliares */
volatile int n_threads_dormindo = 0;
volatile int thr0_ja_executou = 0;
#define N 15
pthread_cond_t cond;
pthread_mutex_t mutex;

void *thread_i(void* v) {
  int i = *(int*) v;
  sleep((random() % 10)/10);
    pthread_mutex_lock(&mutex);
  if (!thr0_ja_executou) {
    printf("Thread %2d vai dormir... \n", i);
    n_threads_dormindo++;
    pthread_cond_wait(&cond, &mutex);
    n_threads_dormindo--;
    printf("Thread %2d acordou. \n", i);
  }
  else
    printf("Thread %2d não vai dormir.\n", i);
  sleep(0.5);
  pthread_mutex_unlock(&mutex);
    return NULL;
}
```

```
void *thread_0(void* v) {
  int j;
  sleep(1);
  pthread_mutex_lock(&mutex);
  for (j = n_threads_dormindo; j > 0; j--) {
    pthread_cond_signal(&cond);
  }
  thr0_ja_executou = 1;
  printf("Thread 0 já mandou cond_signal
     para todas.\n");
  sleep(1);
  pthread_mutex_unlock(&mutex);
  return NULL;
}
int main() {
  pthread_t thr[N];
  int i;
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&cond, NULL);
  for (i = 1; i < N; i++)
    pthread_create(&thr[i], NULL, thread_i,
     (void *) &i);
  pthread_create(&thr[0], NULL, thread_0,
     NULL);
  for (i = 1; i < N; i++)
    pthread_join(thr[i], NULL);
  pthread_join(thr[0], NULL);
  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&cond);
  return 0;
}
```

Exemplo: barreira.c

```
#define N 4
pthread_barrier_t mybarrier;

void* threadFn(void *id_ptr) {
  int thread_id = *(int*)id_ptr;
  int wait_sec = 1 + rand() % 5;
  printf("thread %d: Tempo de espera
    %ds.\n", thread_id, wait_sec);
  sleep(wait_sec);
  printf("thread %d: Pronto\n",
    thread_id);
  pthread_barrier_wait(&mybarrier);
  printf("thread %d: Saindo\n",
    thread_id);
  return NULL;
}
```

```
int main() {
  int I;  pthread_t ids[N];  int short_ids[N];
    srand(time(NULL));
  pthread_barrier_init(&mybarrier, NULL, N +
    1);
  for (i=0; i < N; i++) {
    short_ids[i] = i;
    pthread_create(&ids[i], NULL, threadFn,
     &short_ids[i]);
  }

  pthread_barrier_wait(&mybarrier);
  for (i=0; i < N; i++) {
    pthread_join(ids[i], NULL);
  }
  pthread_barrier_destroy(&mybarrier);

  return 0;
}
```

IPC via Memória Compartilhada

- Uma região de memória é designada como compartilhada, permitindo que múltiplos processos leiam e escrevam nessa área comum
  - Elimina a necessidade de troca de mensagens via canais mais lentos como pipes ou sockets.
- Isso reduz a sobrecarga de CPU e aumenta a velocidade da comunicação interprocessual, tornando-a ideal para aplicações que requerem alta performance e baixa latência, como sistemas em tempo real e servidores de alto desempenho.
- Contudo, é crucial gerenciar a sincronização entre os processos para evitar condições de corrida e garantir a consistência dos dados.

sh1.c

```
#define SHMSZ 27 /* Será arredondado para um múltiplo de PAGE_SIZE */
char str_global[10];

int main() {
  int shmid;
  key_t key;
  char *shm;

  /* Chave arbitrária para o segmento compartilhado */
  key = 5677;

  /* Criação do segmento de memória e obtenção do seu identificador. */
  if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
    perror("shmget");
    exit(1);
  }

  /* Segmento é associado ao espaço de endereçamento */
  if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
  }

  printf("str_local  = %p\n", str_local);
  printf("shm        = %p\n", shm);
  printf("str_global = %p\n", str_global);
  printf("main       = %p\n", main);

  return 0;
}
```

sh_fork.c

```
#define SHMSZ 2
int main() {
  int shmid; key_t key; char *shm; int i=0;
 key = 5677;  /* Chave arbitrária para o segmento compartilhado */

  /* Criação do segmento de memória e obtenção do seu identificador. */
  if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
    perror("shmget");
    exit(1);
  }
  /* Segmento é associado ao espaço de endereçamento */
  if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
  }

  shm[0] = '\0';  shm[1] = '\0';

  if (fork() != 0) {
    shm[0] = '*';  i=1;
    printf("Pai (%d) Esperando o filho\n", getpid());
    while (shm[1] != '*')
      sleep(1);
  }
  else {
    sleep(2);
    shm[1] = '*'; i=2;
    printf("Filho (%d) Esperando o pai\n", getpid());
    while (shm[0] != '*')
      sleep(1);

  }
  printf("%d: i = %d \n",getpid(),i);
  shmctl(shmid, IPC_RMID, NULL);
  shmdt(shm);

  return 0;
}
```

sh_client.c e sh_server.c

```
#define SHMSZ 27
char str_global[10];

int main() {
  int shmid;
  key_t key;
  char str_local[10];
  char *shm;

  /* Chave arbitrária para o compartilhado */
  key = 5679;

  /* Criação do segmento de memória e obtenção do
   seu identificador. */
  if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
    perror("shmget");
    exit(1);
  }

  /* Tentativa de associação próximo à área. */
      if ((shm = shmat(shmid, str_local-0x100000,
   SHM_RND)) == (char *) -1) {
    printf("shmat at %p\n", str_local-0x100000);
    perror("shmat");
    exit(1);
  }

  printf("Processo cliente: %s\n", shm);
  printf("shm  = %p\n", shm);
  shmctl(shmid, IPC_RMID, NULL);
  shmdt(shm);

  return 0;
}
```

```
#define SHMSZ 27
char str_global[10];

int main() {
  int shmid;
  key_t key;
  char *shm;
  char c, *s;

  /* Chave arbitrária para o segmento compartilhado */
  key = 5679;

   /* Criação do segmento de memória e obtenção do seu
   identificador. */
  if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
    perror("shmget");
    exit(1);
  }

  /* Tentativa de associação próximo à área de dados. */
   if ((shm = shmat(shmid, str_global+0x10000, SHM_RND))
   == (char *) -1) {
    printf("shmat at %p\n", str_global+0x1000);
    perror("shmat");
    }
  /* Preenche o segmento com o alfabeto */
  s = shm;
  for (c = 'a'; c <= 'z'; c++)
    *s++ = c;
  *s = '\0';

  printf("Processo servidor: %s\n", shm);
  printf("shm  = %p\n",shm);
  return 0;
}
```

FIM