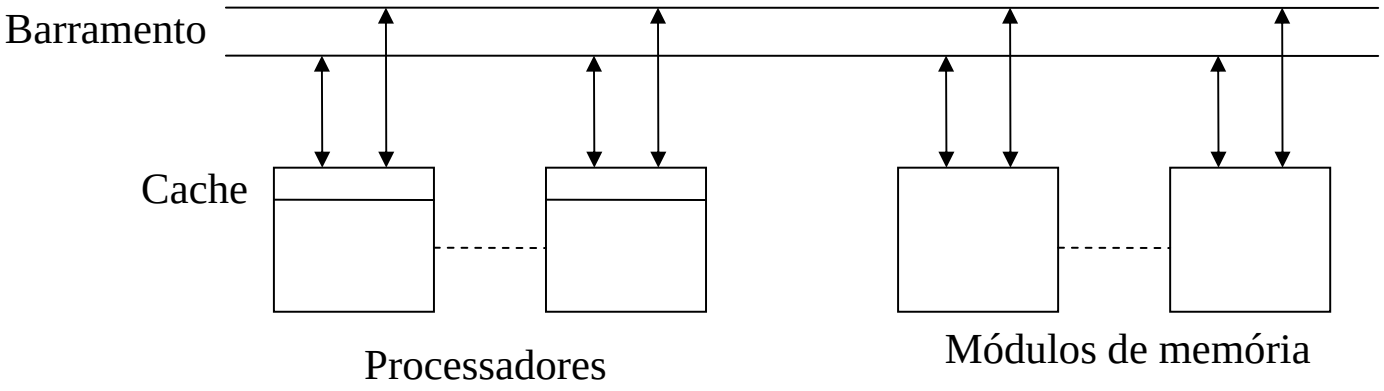
Memória Compartilhada

Programação com memória compartilhada

- **Nos sistemas multiprocessadores com memória compartilhada, qualquer localização da memória pode ser acessada por qualquer processador**
- **Existe um espaço de endereçamento único, ou seja, cada localização de memória possui um único endereço dentro de uma extensão única de endereços**

Arquitetura com barramento único



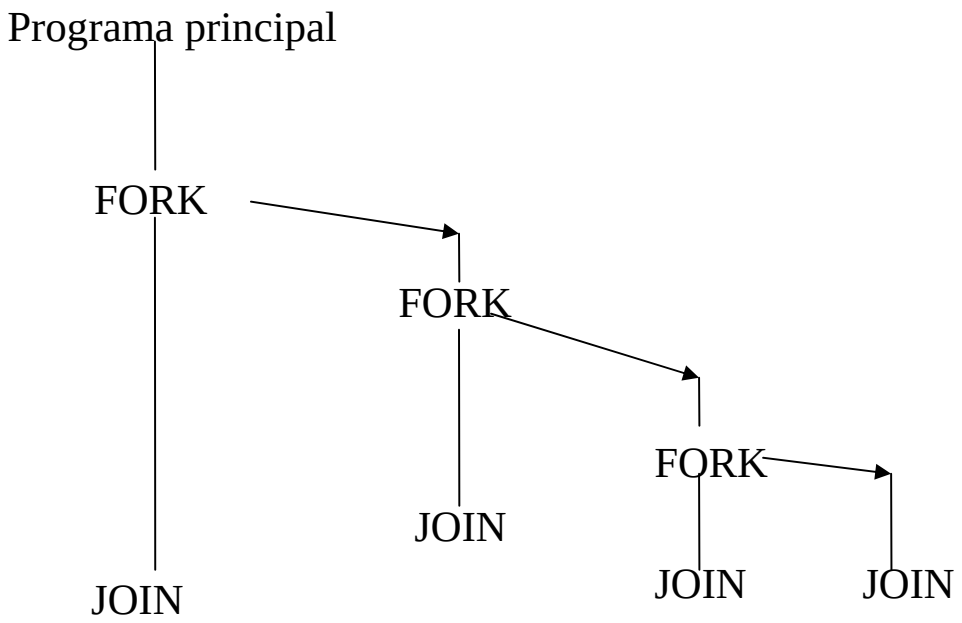
Alternativas para programação

- **Utilizar uma nova linguagem de programação**
- **Modificar uma linguagem seqüencial existente**
- **Utilizar uma linguagem seqüencial existente com rotinas de bibliotecas**
- **Processos UNIX**
- **Threads (Pthreads, Java, ...)**
- **Utilizar uma linguagem de programação seqüencial e deixar a cargo de um compilador paralelizador a geração de um código paralelo executável**

Algumas linguagens de programação paralelas

Linguagem	Criador/data	Comentários
Ada	Depto. de defesa americano	Nova linguagem
C*	Thinking Machines, 1987	Extensão a C para sistemas SIMD
Concurrent C	Gehani e Roome, 1989	Extensão a C
Fortran F	Foz et al., 1990	Extensão a Fortran para programação por paralelismo de dados
Modula-P	Braünl, 1986	Extensão a Modula 2
Pascal concorrente	Brinch Hansen, 1975	Extensão ao Pascal

Criação de processos concorrentes utilizando a construção *fork-join*



Processos UNIX

- **A chamada do UNIX *fork*() cria um novo processo, denominado processo filho**
- **O processo filho é uma cópia exata do processo que chama a rotina *fork*(), sendo a única diferença um identificador de processo diferente**
- **Esse processo possui uma cópia das variáveis do processo pai inicialmente com os mesmos valores do pai**
- **O processo filho inicia a execução no ponto da chamada da rotina**
- **Caso a rotina seja executada com sucesso, o valor 0 é retornado ao processo filho e o identificador do processo filho é retornado ao processo pai**

Processos UNIX

- **Os processos são “ligados” novamente através das chamadas ao sistema:**
  - wait(status): atrasa a execução do chamador até receber um sinal ou um dos seus processos filhos terminar ou parar
  - exit(status): termina o processo
- **Criação de um único processo filho**

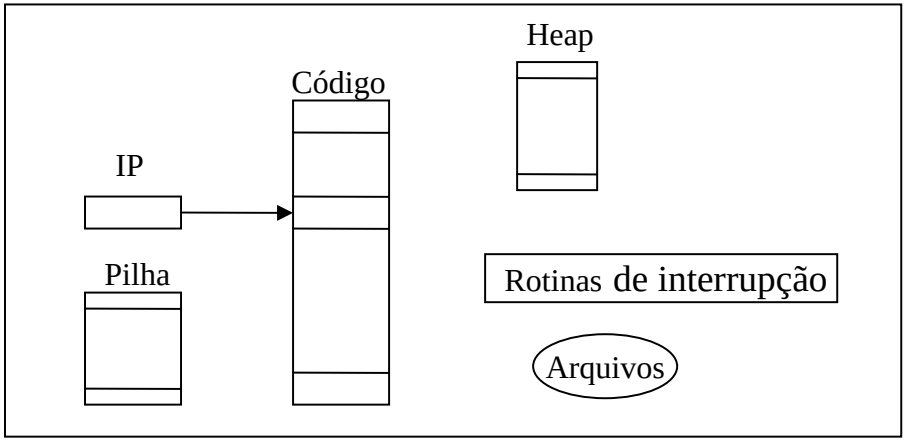
```
pid  =  fork();
código a ser executado pelo pai e filho
if (pid == 0) exit (0); else wait (0)
```

- **Filho executando código diferente**

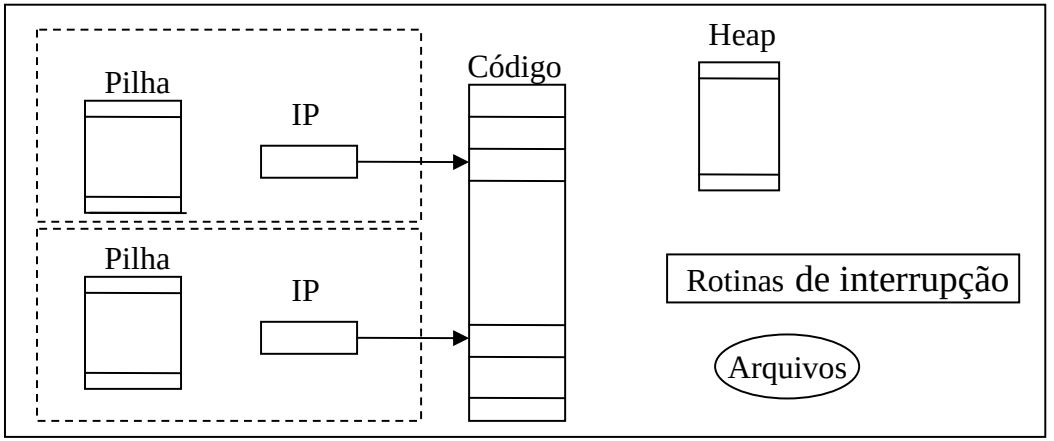
```
pid = fork();
if (pid == 0) {
   código a ser executado pelo filho
} else {
        código a ser executado pelo pai
}
if (pid == 0) exit (0); else wait (0);
```

Threads

Processos: programas completamente separados com suas próprias variáveis, pilha e alocação de memória

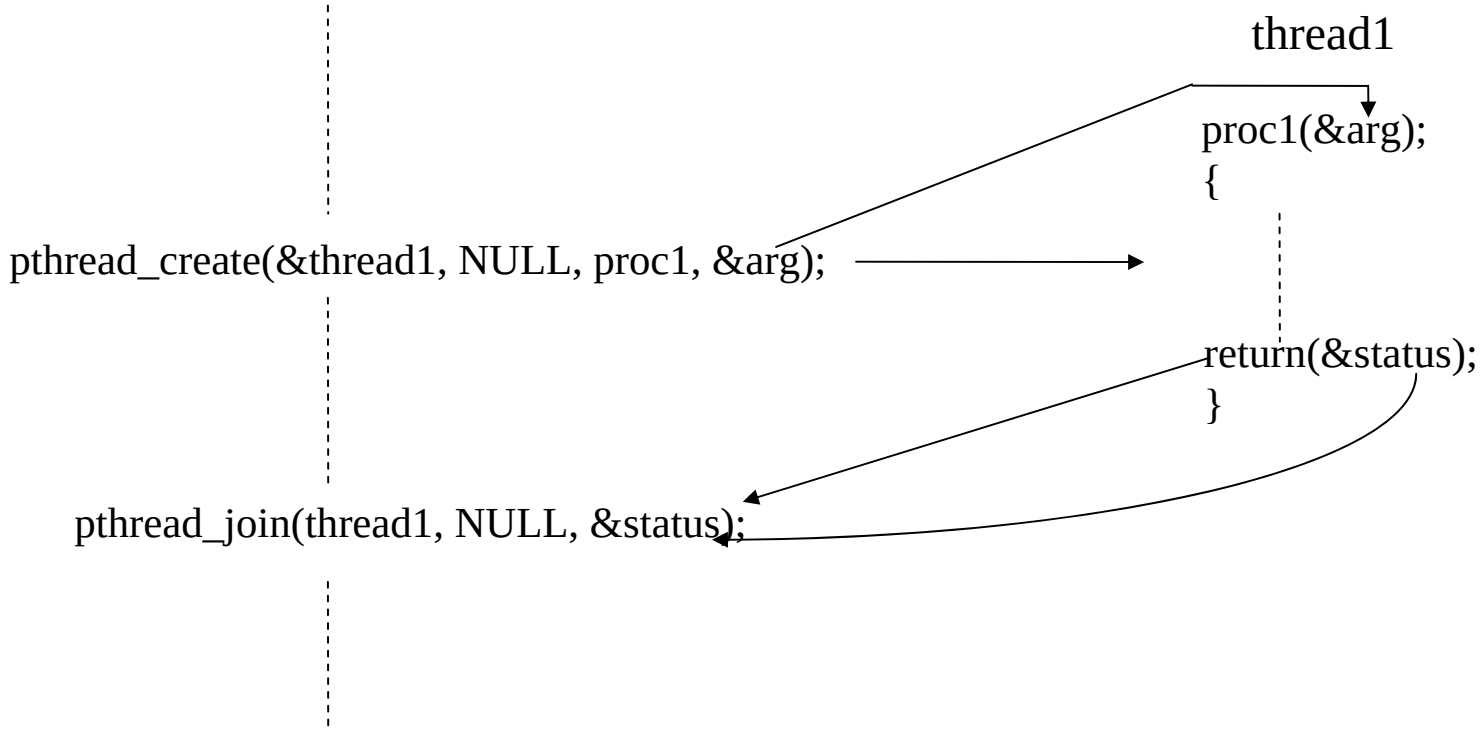


Threads: as rotinas compartilham o mesmo espaço de memória e variáveis globais



Pthreads

Interface portável do sistema operacional, POSIX, IEEE



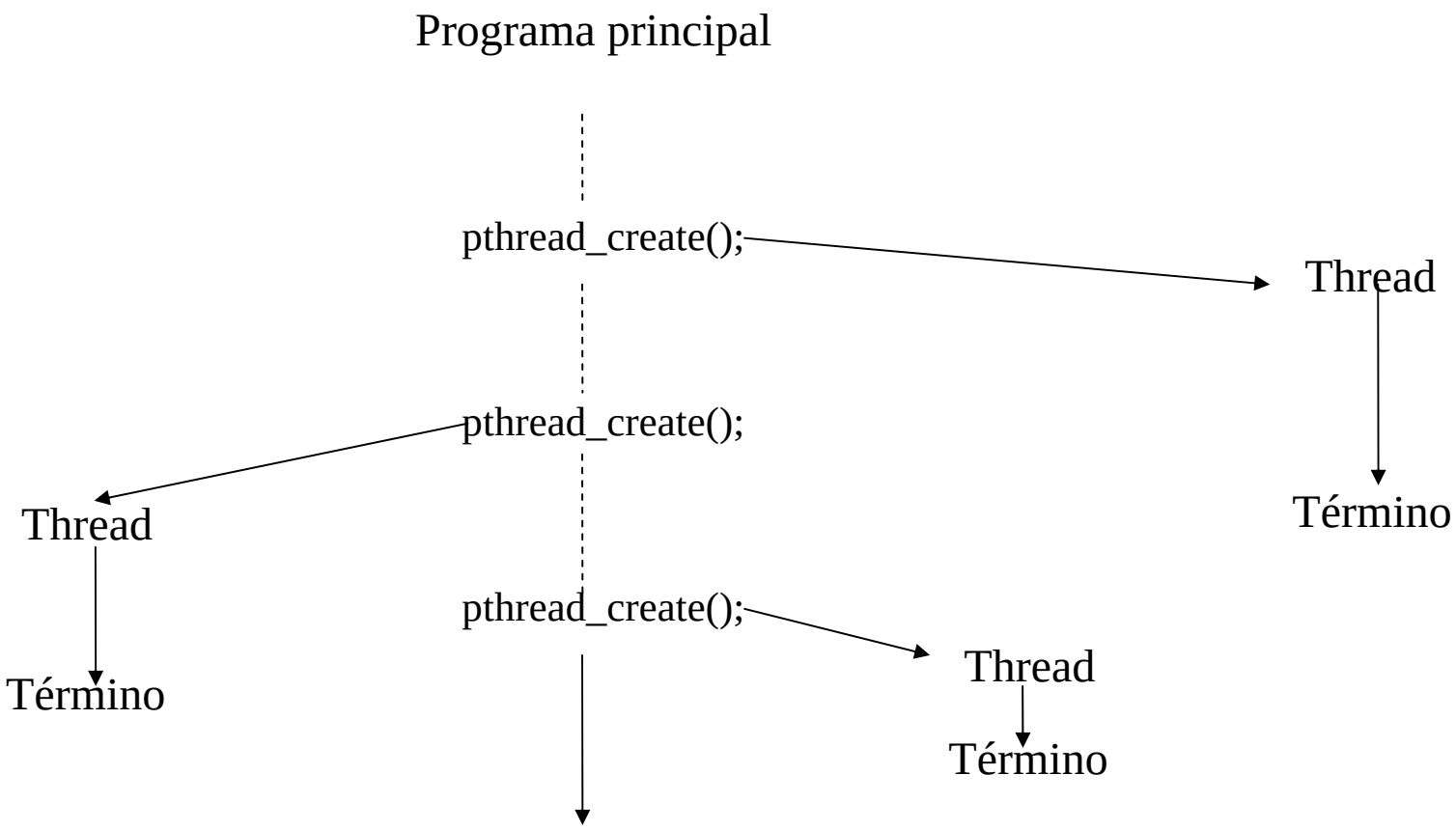
Barreira para Pthreads

- **A rotina *pthread_join*() espera pelo término de uma thread específica**
- **Para criar uma barreira para esperar pelo término de todas as threads, pode-se repetir a chamada da rotina:**

```
for (i = 0; i < n; i++)
    pthread_create(&thread[i], NULL, (void *) slave, (void *) &arg);
      .
      .
for (i = 0; i < n; i++)
    pthread_join(thread[i], NULL);
```

Threads desunidas

Uma thread não precisa saber do término de uma outra por ela criada, então não executa-se a operação de união



Ordem de execução das instruções

- **As execuções das instruções dos processos/threads individuais podem ser entrelaçadas no tempo**
- **Exemplo:**

Processo 1	Processo 2
Instrução 1.1	Instrução 2.1
Instrução 1.2	Instrução 2.2
Instrução 1.3	Instrução 2.3

Possível ordem de execução no tempo:
Instrução 1.1
Instrução 1.2
Instrução 2.1
Instrução 1.3
Instrução 2.2
Instrução 2.3

Ordem de execução das instruções

- **Se dois processos devem imprimir mensagens, por exemplo, as mensagens podem aparecer em ordens diferentes dependendo do escalonamento dos processos que chamam a rotina de impressão**
- **Os caracteres individuais de cada mensagem podem aparecer uns inseridos entre outros, caso as instruções de máquina das instâncias da rotina de impressão sejam divididas em passos menores que possam ser interrompidos**

Otimizações do compilador/processador

- **O compilador ou processador pode reordenar as instruções para tentar otimizar a execução**
- **Exemplo:**
  - As instruções abaixo

    a = b + 5;

    x = y + 4;

    podem ser executadas em ordem contrária:

    x = y + 4;

    a = b + 5;

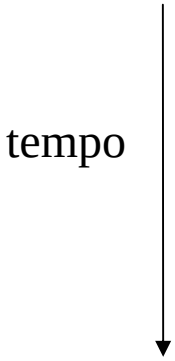
    e o programa continua logicamente correto

Rotinas seguras para threads

- **As chamadas ao sistema ou rotinas de bibliotecas são denominadas seguras para threads, quando elas podem ser acionadas por várias threads ao mesmo tempo e sempre produzem resultados corretos**
- **Rotinas padrão de E/S: imprimem mensagens sem permitir interrupção entre impressão de caracteres**
- **Rotinas que acessam dados compartilhados estáticos requerem cuidados especiais para serem seguras para threads**
- **Pode-se forçar a execução da rotina somente por uma thread de cada vez, ineficiente**

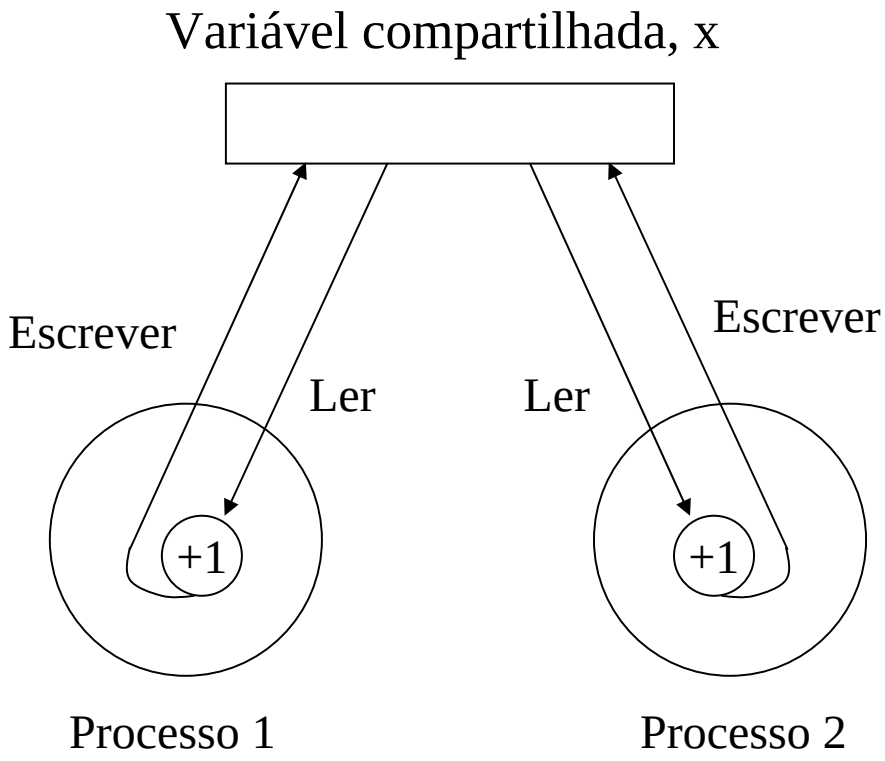
Acessando dados compartilhados

- **Considere dois processos que devem ler o valor de uma variável *x*, somar 1 a esse valor e escrever o novo valor de volta na variável *x***



Instrução	Processo 1	Processo 2
x = x + 1	ler x	ler x
	calcular x + 1	calcular x + 1
	escrever x	escrever x

Conflito em acesso a dados compartilhados



Seção crítica

- **Devem existir mecanismos para assegurar que somente um processo possa acessar um determinado recurso durante um certo tempo**
- **Estabelecem-se sessões de código que envolvem o recurso, denominadas seções críticas, e organiza-se o acesso a elas de tal modo que somente uma delas pode ser executada por um processo de cada vez**
- **Um processo impede que outros processos acessem as seções críticas que possui um determinado recurso compartilhado quando ele estiver acessando**
- **Quando um processo finaliza a execução da seção crítica, outro processo pode executá-la**
- **Mecanismo conhecido como exclusão mútua**

Lock

- **Mecanismo mais simples de assegurar exclusão mútua para acesso a seções críticas**
- **O lock é uma variável de 1 bit que possui o valor 1 quando existe algum processo na seção crítica e 0 quando nenhum processo está na seção crítica**
- **Um processo que chega a uma porta da seção crítica e a acha aberta pode entrar, trancando-a para prevenir que nenhum outro processo entre**
- **Assim que o processo finaliza a execução da seção crítica, ele destranca a porta e sai**

Spin lock

```
while (lock == 1) do_nothing;
lock = 1;
        .
    seção crítica
        .
lock=0;
```

Processo 1 Processo 2

```
while (lock == 1) do _nothing;      while (lock == 1) do_nothing;
lock =1;
```

seção crítica

```
lock=0;
                    lock = 1;
```

seção crítica

```
                    lock = 0;
```

Rotinas de lock para Pthreads

- **Os locks são implementados com variáveis lock mutuamente exclusivas, ou variáveis “mutex”**
- **Para utilizá-la, deve-se declará-la com o tipo *pthread_mutex_t* e inicializá-la:**

```
pthread_mutex_t mutex1;
    .
    .
pthread_mutex_init(&mutex1, NULL);
```

- **Uma variável mutex pode ser destruída utilizando-se a rotina *pthread_mutex_destroy*()**

Rotinas de lock para Pthreads

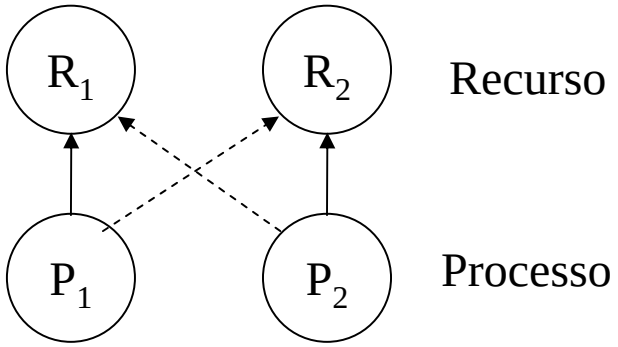
- **Uma seção crítica pode ser protegida utilizando-se *pthread_mutex_lock*() e *pthread_mutex_unlock*()**

```
pthread_mutex_lock(& mutex1);
    .
seção crítica
    .
pthread_mutex_unlock(&mutex1);
```

- **Se uma thread chega em uma rotina de trancamento de uma variável mutex e a encontra trancada, ela espera até que seja destrancada**
- **Se mais de uma thread está esperando o destrancamento, o sistema escolhe aquela que poderá prosseguir**
- **Somente a thread que tranca uma variável pode destrancá-la**

Deadlock

- **Pode ocorrer com dois processos quando um deles precisa de um recurso que está com outro e esse precisa de um recurso que está com o primeiro**



Rotina para evitar deadlock em Pthreads

- **Pthreads oferece uma rotina para testar se uma variável está trancada sem bloquear a thread**
- **pthread_mutex_trylock()**
  - tranca uma variável mutex que esteja destrancada e retorna o valor 0, ou retorna o valor EBUSY caso a variável esteja trancada

Semáforos

- **Um semáforo *s* é um inteiro positivo (incluindo o 0) operado por duas operações denominadas P e V**
- **Operação P(s)**
  - espera até que *s* seja maior que 0, a decrementa de 1 e permite que o processo continue a sua execução
- **Operação V(s)**
  - incrementa *s* de 1 para liberar algum processo que esteja esperando
- **As operações P e V são executadas de forma indivisível**
- **Mecanismo de espera implícito nas operações**
- **Os processos atrasados por P(s) são mantidos parados até serem liberados por uma operação V(s) sobre o mesmo semáforo**

Exclusão mútua de seções críticas

- **Pode ser implementada através de um semáforo binário, que só assume os valores 0 e 1**
- **Funciona como uma variável lock mas as operações P e V incluem um mecanismo de escalonamento de processos**
- **O semáforo é inicializado com o valor 1, indicando que nenhum processo está na sua seção crítica associada ao semáforo**

Semáforo geral

- **Pode ter qualquer valor positivo**
- **Pode fornecer, por exemplo, um modo de registrar o número de unidades do recurso disponíveis ou não disponíveis e pode ser utilizada para resolver problemas de consumidor/produtor**
- **Rotinas de semáforo existem para processos UNIX**
- **Não existem para Pthreads, somente para extensão de tempo real**

Monitor

- **Um conjunto de procedimentos que fornecem o único método de acessar um recurso compartilhado**
- **Os dados e operações que podem ser executadas sobre os dados são encapsulados em uma estrutura**
- **A leitura e escrita de dados só podem ser feitas utilizando-se procedimentos do monitor, e somente um processo pode utilizar um procedimento de monitor em qualquer instante**
- **Um procedimento de monitor pode ser implementado utilizando-se semáforos**

```
monitor_proc1()
{
   P(monitor_semaphore);
    .
    Corpo do monitor
    .
  V(monitor_semaphore);
  return;
}
```

Variáveis de condição

- **Freqüentemente, uma seção crítica deve ser executada caso exista uma condição global específica, por exemplo, o valor de uma variável chegou a um determinado valor**
- **Utilizando-se locks, a variável global tem que ser freqüentemente examinada dentro da seção crítica**
- **Consome tempo de CPU**
- **Utilizam-se variáveis de condição**

Operações para variáveis de condição

- **Três operações:**
  - wait(cond_var) - espera que ocorra a condição
  - signal(cond_var) - sinaliza que a condição ocorreu
  - status(cond_var) - retorna o número de processos esperando pela condição
- **A rotina wait() libera o semáforo ou lock e pode ser utilizada para permitir que outro processo altere a condição**
- **Quando o processo que chamou a rotina wait() é liberado para seguir a execução, o semáforo ou lock é trancado novamente**

Operações para variáveis de condição (exemplo)

- **Considere um ou mais processos (ou threads) designados a executar uma determinada operação quando um contador *x* chegue a zero**
- **Outro processo ou thread é responsável por decrementar o contador**

```
action()                    contador()
{                           {
  .                         .
    lock();                   lock();
    while (x != 0)                x--;
  wait(s);          ◄·····  if (x == 0) signal (s);
    unlock();                 unlock();
    take_action();                  .
}                           }
```

Variáveis de condição em Pthreads

- **Associadas com uma variável mutex específica**
- **Declarações e inicializações:**

```
pthread_cond_t  cond1;
pthread_mutex_t mutex1;
pthread_cond_init(&cond1, NULL);
pthread_mutex_init(&mutex1, NULL);
```

- **Utilização das rotinas de wait e signal:**

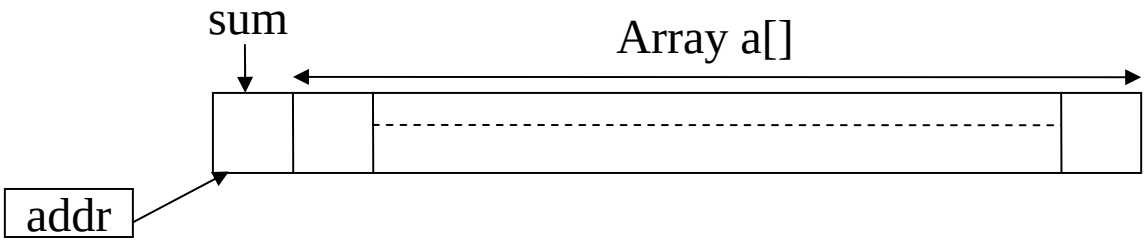
```
action()                              contador()
{                                     {
  .                                     .
    pthread_mutex_lock(&mutex1);          pthread_mutex_lock(&mutex1);
    while (c<> 0)                           c--;
      pthread_cond_wait(cond1, mutex1);←  if (c == 0) pthread_cond_signal(cond1);
    pthread_mutex_unlock(&mutex1);        pthread_mutex_unlock(&mutex1);
    take_action();                          .
}                                     }
```

Exemplo utilizando processos UNIX

- **O cálculo é dividido em duas partes, uma para a soma dos elementos pares e outra para os ímpares**

```
Processo 1                    Processo 2
sum1 = 0                      sum2 = 0
for (i = 0; i < 1000; i = i+2)   for (i = 1; i < 1000;  i= i+2)
    sum1 = sum1 + a[i]               sum2 = sum2 + a[i]
```

- **Cada processo soma o seu resultado (sum1 ou sum2) a uma variável compartilhada sum que acumula o resultado e deve ser ter seu acesso protegido**
- **Estrutura de dados utilizada**



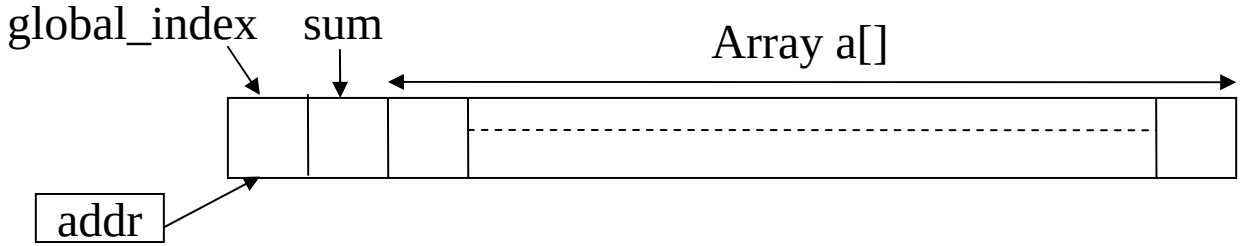
```
#define array_size 1000
extern char *shmat()
void P(int *s);
void V(int *s);
int main()
{
int shmid, s, pid;
char *shm;
int *a, *addr, *sum;
int partial_sum;
int i;
int init_sem_value =1 ;
s = semget(IPC_PRIVATE, 1, (0600 | IPC_CREAT));
if (s == -1) {
   perror(“semget”);
   exit(1);
}
if semctl(s, 0, SETVAL, init_sem_value) < 0) {
   perror(“semctl”);
   exit(1);
}
shmid = shmget(IPC_PRIVATE, (array_size*sizeof(int)+1), IPC_CREAT|0600);
if (shmid == -1)  {
    perror(“shmget”);
    exit (1);
}
shm = shmat(shmid, NULL, 0);
if (shm == (char*)-1) {
   perror(“shmat”);
   exit(1);
}
```

```
addr = (int *) shm;
sum = addr;
addr++;
a = addr;
*sum = 0;
for (i = 0; i < array_size; i++)
    *(a+i) = i+1;
pid = fork();
if (pid ==0)  {
   partial_sum=0;
   for (i=0; i<array_size; i=i+2)
      partial_sum+=*(a+i);
else  {
   partial_sum=0;
   for (i=1; i<array_size; i=i+2)
      partial_sum+=*(a+i);
}
P(&s);
*sum+=partial_sum;
V(&s);
printf("\n pid do processo = %d, soma parcial = %d \n", pid, partial_sum);
if (pid == 0) exit (0); else wait(0);
printf("A soma total é %d \n", *sum);
if (semctl(s, 0, IPC_RMID,1) == -1) {
    perror("semctl");
    exit(1);
}
if (shmctl(shmid, IPC_RMID, NULL) == -1) {
    perror("shmctl");
    exit(1);
}
}
```

```
void P(int *s)
{
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num=0;
   sops->sem_op=-1;
   sops->sem_flg=0;
   if  (semop(*s, sops, 1) < 0)  {
      perror (“semop”);
      exit (1);
   }
   return;
}
void V(int *s)
{
   struct sembuf sembuffer, *sops;
   sops = &sembuffer;
   sops->sem_num=0;
   sops->sem_op=1;
   sops->sem_flg=0;
   if  (semop(*s, sops, 1) < 0)  {
      perror (“semop”);
      exit (1);
   }
   return;
}
```

Exemplo utilizando Pthreads

- **São criadas *n* threads, cada uma obtém os números de uma lista, os soma e coloca o resultado em uma variável compartilhada denominada *sum***
- **A variável compartilhada global_index é utilizada por cada thread para selecionar o próximo elemento de *a***
- **Após a leitura do índice, ele é incrementado para preparar para a leitura do próximo elemento**
- **Estrutura de dados utilizada**



```
#define array_size 1000;
#define no_threads 10;

int a[array_size];
int global_index = 0;
int sum = 0;
pthread_mutex_t mutex1;
void *slave (void  *ignored)
{
  int local_index, partial_sum =0;
  do   {
     pthread_mutex_lock&(mutex1);
    local_index = global_index;
    global_index++;
    pthread_mutex_unlock(&mutex1);
    if (local_index < array_size)
      partial_sum += *(a+local_index);
  } while (local_index < array_size);
  pthread_mutex_lock(mutex1);
  sum+=partial_sum;
  pthread_mutex_unlock(&mutex1);
  return();
}
```

```
main()
{
   int i;
   pthread_t thread[no_threads];
   pthread_mutex_init(&mutex1, NULL);
   for (i = 0; i < array_size; i++)
      a[i] = i+1;
   for (i = 0; i < no_threads; i++)
      if (pthread_create(&thread[i], NULL, slave, NULL) != 0) {
         perror("Pthread_create falhou");
         exit(1); }
  for (i = 0; i < no_threads; i++)
      if (pthread_join(thread[i], NUL) != 0) {
         perror("Pthread_join falhou");
         exit(1); }
printf("A soma é %d \n", sum)
}
```