Programação em Memória Partilhada com Processos

Ricardo Rocha

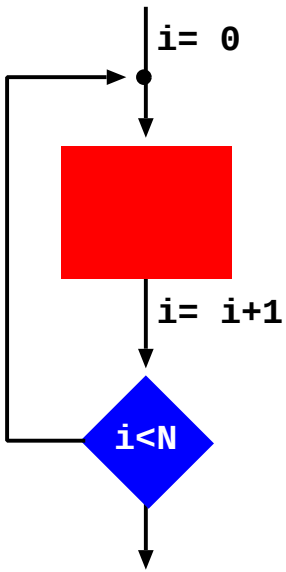
Departamento de Ciência de Computadores
Faculdade de Ciências
Universidade do Porto

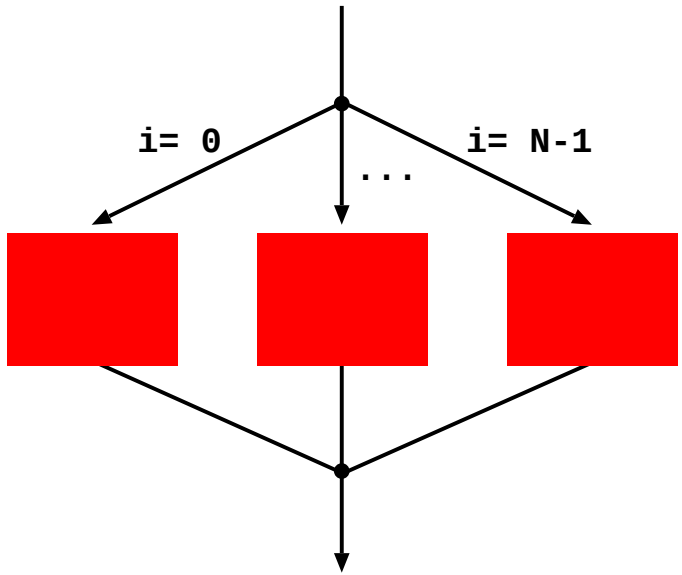
Computação Paralela 2015/2016

Paralelismo nos Dados

Paralelismo nos dados é uma das técnicas mais simples de explorar paralelismo cuja ideia se baseia em **realizar a mesma operação sobre cada componente dos dados**:

- Os dados surgem normalmente sob a forma de vetores/matrizes multidimensionais
- Os ciclos são os principais candidatos a serem paralelizados
- Frequente em problemas científicos e de engenharia

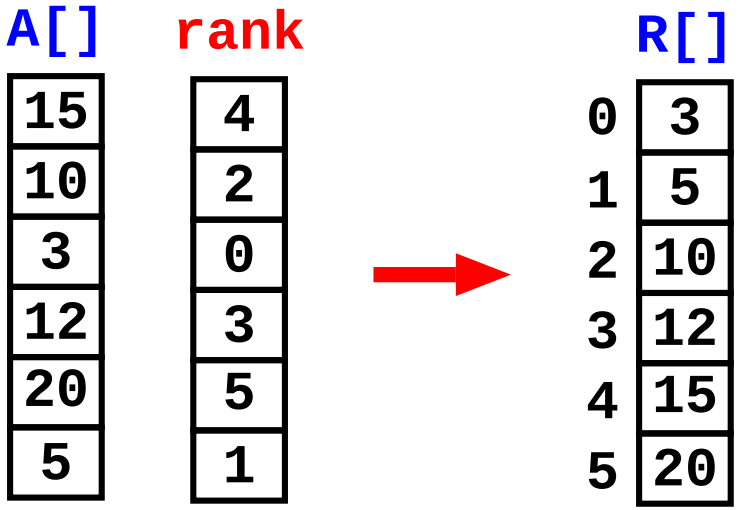




Rank Sort

Dado um vetor `A[N]`, pretende-se construir um vetor `R[N]` com os elementos de `A[N]` ordenados:

- Para cada elemento `A[k]` vamos determinar a sua posição relativa (**rank**) no vetor `R[N]`. Essa posição pode ser obtida calculando o número de elementos em `A[N]` que são menores que `A[k]`
- Como o cálculo da posição relativa de cada elemento é uma tarefa independente, o algoritmo pode ser facilmente paralelizável



Rank Sort

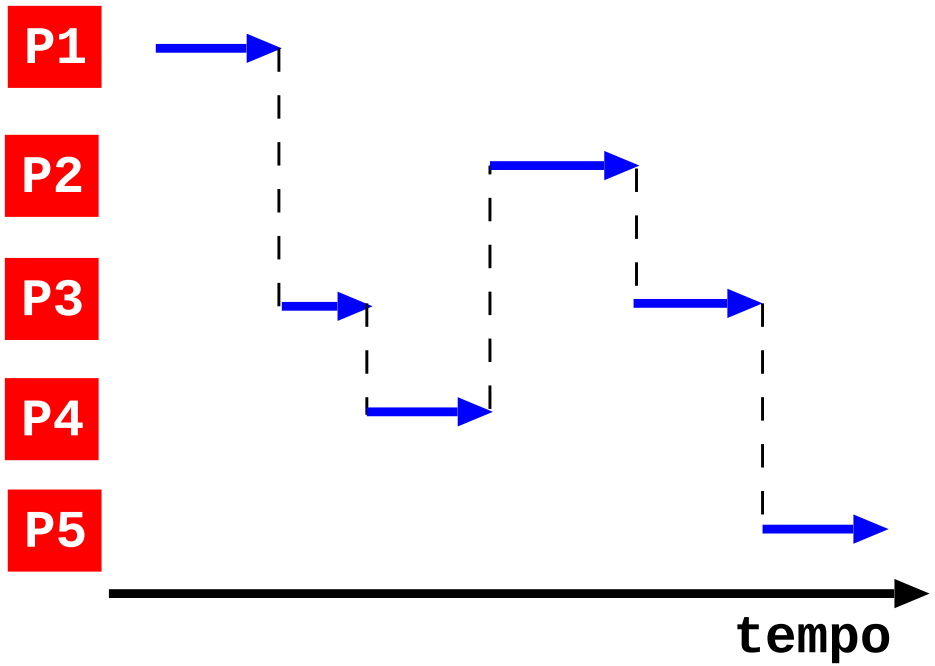
```
int A[N], R[N];

main() {
  ...
  for (k = 0; k < N; k++)
    compute_rank(A[k]);
  ...
}

compute_rank(int elem) {
  int i, rank = 0;
  for (i = 0; i < N; i++)
    if (elem > A[i])
      rank++;
  R[rank] = elem;
}
```

Questão: como paralelizar o algoritmo do rank sort?

Processos

Um processo é a **abstração de um programa em execução**, o que permite que um programa possa ter várias instâncias em execução.

Em máquinas uniprocessador, em cada instante, apenas um processo está em execução. No entanto, vários processos podem ser executados numa determinada fração de tempo, dando ao utilizador a ilusão de paralelismo.

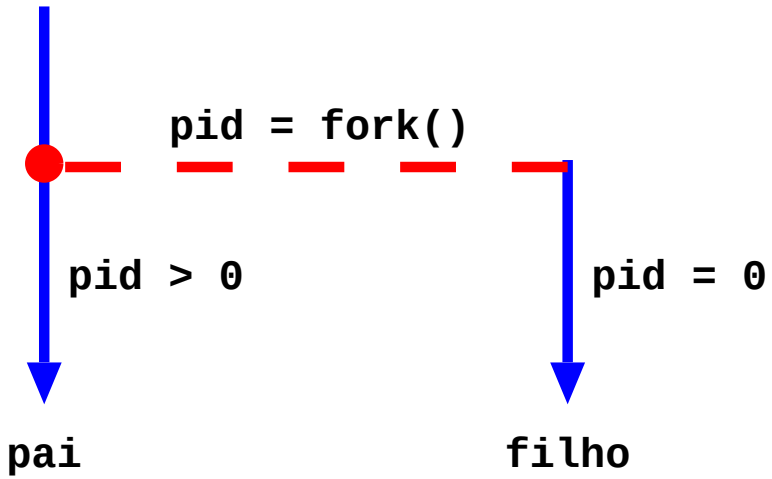


Criação de Processos

```
pid_t fork(void)
```

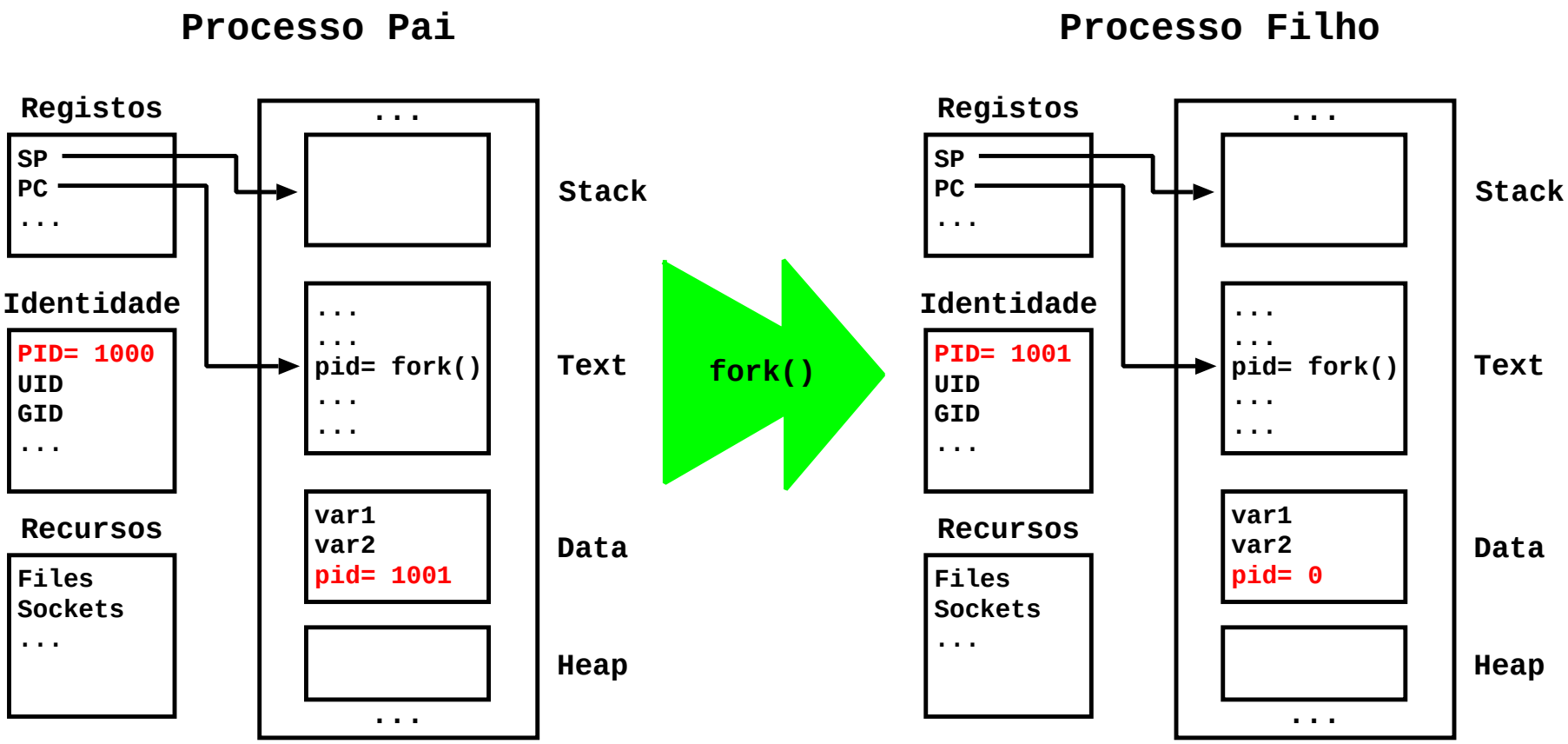
A chamada ao sistema `fork()` permite criar novos processos. Retorna o PID do novo processo (**processo filho**) ao processo que faz a chamada (**processo pai**) e retorna 0 ao processo filho.

Como distinguir a execução do processo filho da do processo pai?




```
pid_t pid;
...
pid = fork();
if (pid == 0) {
  ... // child code after fork
} else {
  ... // parent code after fork
}
// common code after fork
```

Criação de Processos



Parallel Rank Sort (proc-ranksort.c)

```
main() {
  ...
  // each child executes one task
  for (k = 0; k < N; k++)
    if (fork() == 0) {
      compute_rank(A[k]);
      exit(0);
    }
  // parent waits for all children to complete
  for (k = 0; k < N; k++)
    wait(NULL);
  // parent shows result
  for (k = 0; k < N; k++)
    printf("%d ", R[k]);
  printf("\n");
  ...
}
```

Questão: o que é que o programa escreve no final?

Parallel Rank Sort

Lançamos N processos filho em que cada um executa `compute_rank()` para um elemento diferente `A[k]`:

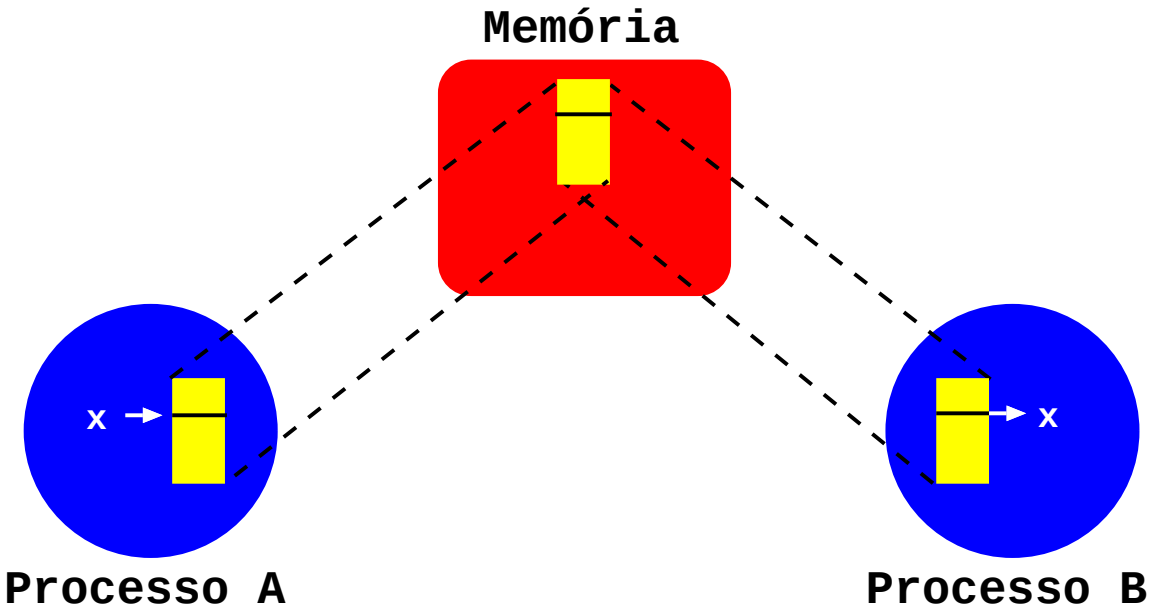
- Cada processo filho herda uma cópia das variáveis do processo pai, mas ao alterar essas variáveis o processo pai não vê as alterações
- As alterações feitas sobre `R[]` não ficam visíveis e o processo pai escreve uma sequência de zeros!

Solução: o vetor `R[]` tem de ser partilhado!

Segmentos de Memória Partilhada

Um dos métodos mais simples de comunicação entre processos (**interprocess communication (IPC)**) é a utilização de segmentos de memória partilhada:

- O segmento é conhecido de ambos os processos e quando um dos processos escreve no segmento, o outro também vê a alteração
- O acesso a segmentos de memória partilhada é tão eficaz como a segmentos não partilhados e a sua manipulação é idêntica



Segmentos de Memória Partilhada

Sequência de utilização de um segmento de memória partilhada:

- Inicialmente, os processos devem **alocar o segmento**
- Em seguida, cada processo, deve **mapear o segmento** num endereço de memória de modo a poder utilizá-lo
- Após a sua utilização, cada processo **liberta o mapeamento feito**
- Por fim, um dos processos **remove o segmento**

Alocar um Segmento de Memória Partilhada

```
int shmget(key_t key, int size, int flags)
```

`shmget()` aloca um novo segmento de memória partilhada e retorna o id do segmento. Caso não seja possível alocar o segmento retorna -1.

- `key` é a chave que permite identificar o segmento pretendido. Outros processos podem aceder ao mesmo segmento se indicarem a mesma chave (`IPC_PRIVATE` garante que um novo segmento é atribuído)
- `size` é o tamanho pretendido para o segmento arredondado para um múltiplo do tamanho das páginas do sistema (normalmente 4KB – `getpagesize()` para obter o valor exato)
- `flags` permite especificar opções de alocação: `IPC_CREAT` indica que um novo segmento deve ser criado (caso não exista); `IPC_EXCL` indica que o segmento deve ser exclusivo (falha caso contrário); `S_IRUSR`, `S_IWUSR`, `S_IROTH` e `S_IWOTH` indicam permissões de leitura e escrita

Mapear um Segmento de Memória Partilhada

```
void *shmat(int shmid, void *addr, int flags)
```

`shmat()` permite mapear um segmento de memória partilhada a partir de um endereço de memória no espaço de endereçamento do processo. Retorna o endereço de memória no qual o segmento foi mapeado. Caso não seja possível mapear o segmento retorna -1.

- `shmid` é o inteiro que identifica o segmento (obtido com `shmget()`)
- `addr` é o endereço de memória pretendido (deve ser um múltiplo do tamanho das páginas do sistema) ou `NULL`, caso se pretenda que seja o sistema a escolher o endereço
- `flags` permite especificar opções de mapeamento: `SHM_RDONLY` indica que o segmento é apenas de leitura

Libertar um Segmento de Memória Partilhada

```
int shmdt(void *addr)
```

`shmdt()` liberta o mapeamento feito e o segmento partilhado correspondente deixa de ficar associado a um endereço de memória (o sistema decrementa em uma unidade o número de mapeamentos do segmento). Retorna 0 se OK, -1 se erro.

- `addr` é o endereço inicial do segmento de memória a libertar

Remover um Segmento de Memória Partilhada

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

`shmctl()` remove o segmento de memória partilhada e não permite mais mapeamentos (só é realmente removido quando o número de mapeamentos for zero). Retorna 0 se OK, -1 se erro.

- `shmid` é o inteiro que identifica o segmento
- `cmd` deve ser `IPC_RMID`
- `buf` deve ser `NULL`

O número de segmentos de memória partilhada disponíveis é limitado. O fim de um processo liberta automaticamente o mapeamento mas não remove o segmento. `shmctl()` deve ser explicitamente invocado por um dos processos.

- Comando `ipcs` permiter ver os segmentos que estão a ser usados
- Comando `ipcrm` permitir remover um segmento

Sequência de Utilização Básica

```
int shmid, shmsize;
char *shared_memory;
...
shmsize = getpagesize();
shmid = shmget(IPC_PRIVATE, shmsize, S_IRUSR | S_IWUSR);
shared_memory = (char *) shmat(shmid, NULL, 0);
...
sprintf(shared_memory, "Hello World!");
...
shmdt(shared_memory);
shmctl(shmid, IPC_RMID, NULL);
```

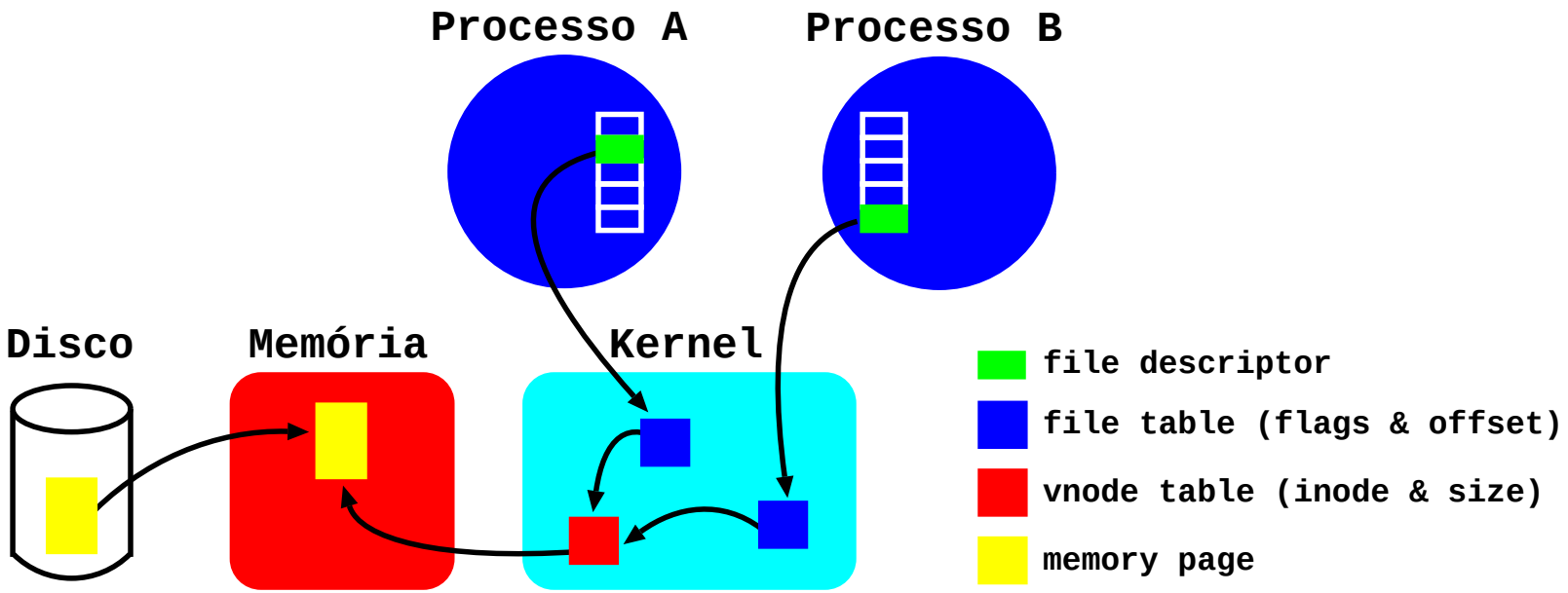
Parallel Rank Sort (proc-rankshm.c)

```
int A[N], *R;

main() {
  ...
  // allocate and map a shared segment for R[]
  shmid = shmget(IPC_PRIVATE, N * sizeof(int), S_IRUSR | S_IWUSR);
  R = (int *) shmat(shmid, NULL, 0);
  // each child executes one task
  for (k = 0; k < N; k++)
    if (fork() == 0) {
      compute_rank(A[k]);
      exit(0);
    }
  for (k = 0; k < N; k++) wait(NULL);
  for (k = 0; k < N; k++) printf("%d ", R[k]); printf("\n");
  // free and remove shared segment
  shmdt(R);
  shmctl(shmid, IPC_RMID, NULL);
}
```

Mapeamento de Ficheiros em Memória

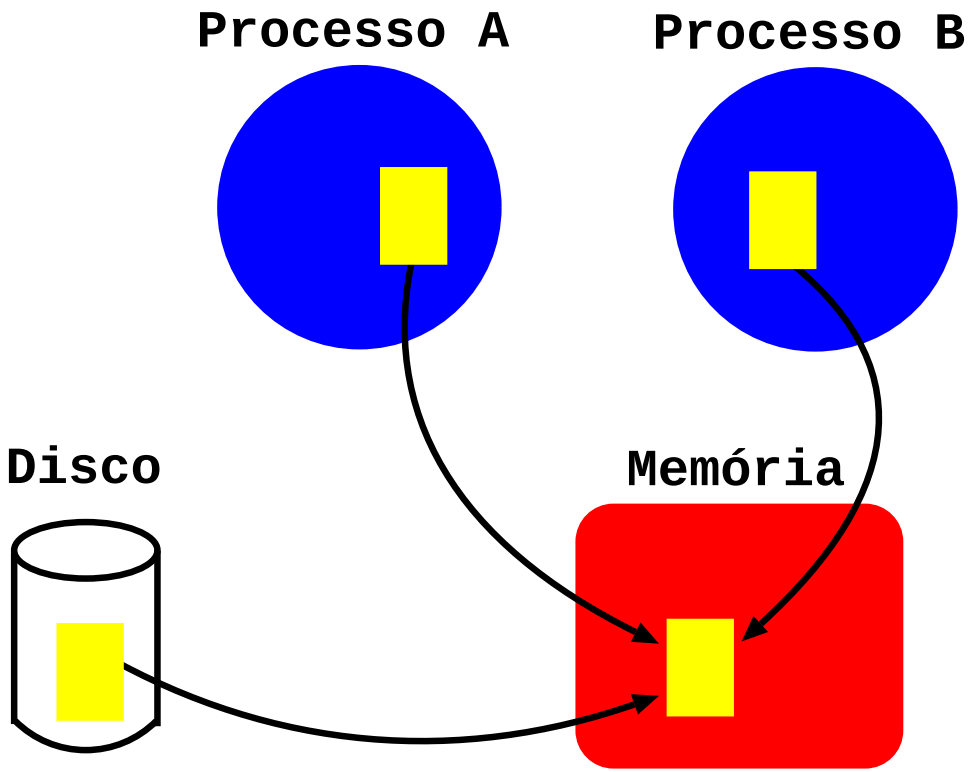
A comunicação entre processos utilizando memória partilhada também pode ser conseguida via **ficheiros partilhados**.

- O acesso a ficheiros é habitualmente conseguido por uso das funções `open()`, `read()`, `write()`, `lseek()` e `close()`
- A atomicidade na escrita e leitura de um ficheiro é garantida pelas operações de `read()` e `write()` que sincronizam na estrutura de dados **vnode** associada ao ficheiro



Mapeamento de Ficheiros em Memória

O interessante é permitir que um processo possa mapear regiões de um ficheiro diretamente no seu espaço de endereçamento de forma a que as operações de leitura e escrita sejam transparentes.



Mapeamento de Ficheiros em Memória

Sequência de utilização para mapear um ficheiro num espaço de memória:

- Inicialmente, os processos devem **obter o descritor do ficheiro** a mapear
- Em seguida, cada processo, deve **mapear o ficheiro** num endereço de memória de modo a poder utilizá-lo
- Após a sua utilização, cada processo **liberta o mapeamento feito**

Mapear uma Região de um Ficheiro em Memória

```
void *mmap(void *start, size_t length, int prot, int flags,
  int fd, off_t offset)
```

`mmap()` permite mapear uma região de um ficheiro a partir de um endereço de memória no espaço de endereçamento do processo. Retorna o endereço de memória no qual a região foi mapeada. Caso não seja possível mapear a região retorna -1.

- `start` é o endereço de memória pretendido (deve ser um múltiplo do tamanho das páginas do sistema) ou `NULL`, caso se pretenda que seja o sistema a escolher o endereço
- `length` é o tamanho a mapear em bytes
- `prot` permite especificar opções de mapeamento: `PROT_READ` e `PROT_WRITE` indicam permissões de leitura e escrita

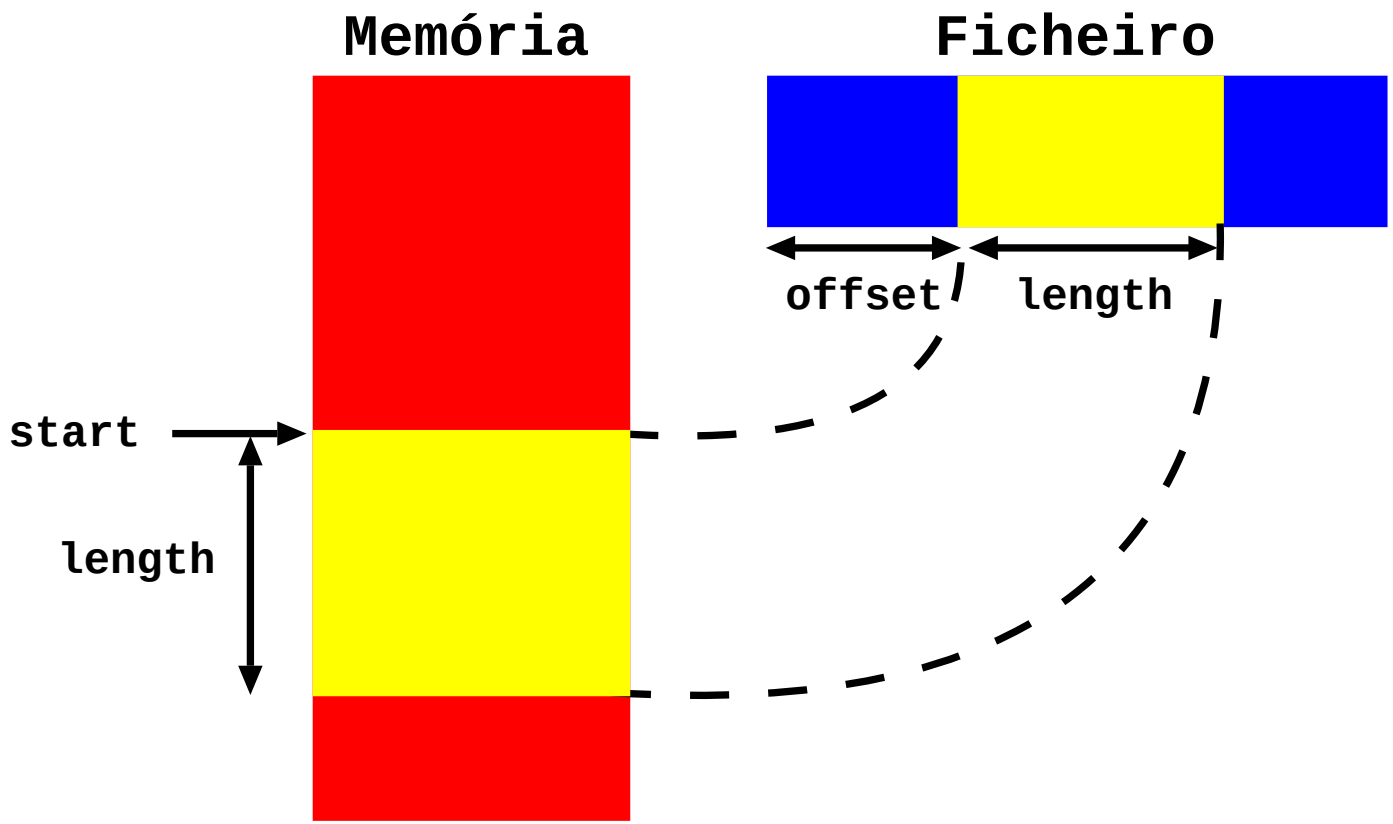
Mapear uma Região de um Ficheiro em Memória

```
void *mmap(void *start, size_t length, int prot, int flags,
  int fd, off_t offset)
```

- `flags` permite especificar atributos do mapeamento: `MAP_FIXED` obriga a usar necessariamente `start` para mapear a região; `MAP_SHARED` indica que as operações de escrita modificam o ficheiro; `MAP_PRIVATE` indica que as operações de escrita não se refletem no ficheiro (normalmente utilizada para debugging)
- `fd` é o descritor do ficheiro a mapear
- `offset` é o deslocamento no ficheiro da região a mapear (deve ser um múltiplo do tamanho das páginas do sistema)

Mapear uma Região de um Ficheiro em Memória

```
void *mmap(void *start, size_t length, int prot, int flags,
  int fd, off_t offset)
```



Libertar uma Região de Memória Mapeada

```
int munmap(void *start, size_t length)
```

`munmap()` liberta o mapeamento feito e a região de memória correspondente deixa de ficar associada a um endereço de memória. Retorna 0 se OK, -1 se erro.

- `start` é o endereço inicial da região de memória a libertar
- `length` é o tamanho a libertar

Sequência de Utilização Básica

```
int fd, mapsize;
void *mapped_memory;
...
mapsize = getpagesize();
fd = open("mapfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
lseek(fd, mapsize, SEEK_SET);
write(fd, "", 1);
mapped_memory = mmap(NULL, mapsize, PROT_READ | PROT_WRITE,
                     MAP_SHARED, fd, 0);
...
sprintf(mapped_memory, "Hello World!");
...
munmap(mapped_memory, mapsize);
```

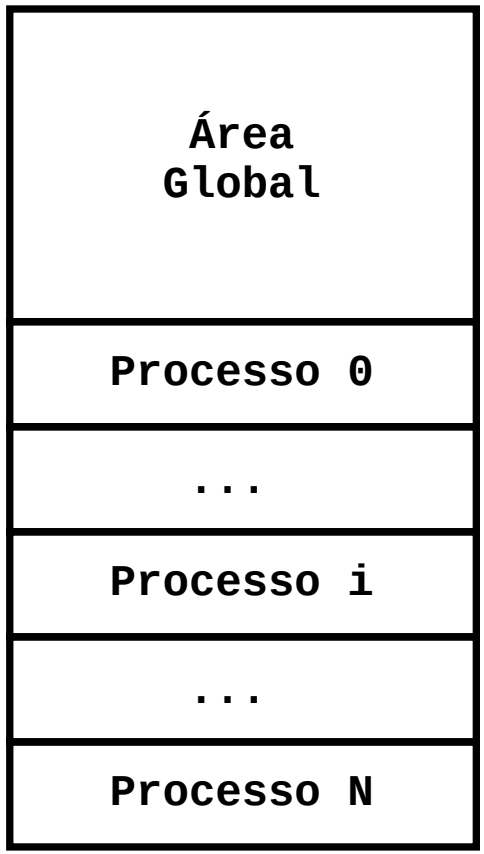
Parallel Rank Sort (proc-rankmmap.c)

```
int A[N], *R;

main() {
  ...
  // map a file into a shared memory region for R[]
  fd = open("mapfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  lseek(fd, N * sizeof(int), SEEK_SET);
  write(fd, "", 1);
  R = (int *) mmap(NULL, N * sizeof(int), PROT_READ | PROT_WRITE,
                   MAP_SHARED, fd, 0);
  // each child executes one task
  for (k = 0; k < N; k++)
    if (fork() == 0) {
      compute_rank(A[k]);
      exit(0);
    }
  for (k = 0; k < N; k++) wait(NULL);
  for (k = 0; k < N; k++) printf("%d\n", R[k]); printf("\n");
  // unmap shared memory region
  munmap(R, N * sizeof(int));
}
```

Técnicas Avançadas de Mapeamento de Memória

Considere um mapeamento de memória partilhada tal como na figura ao lado:

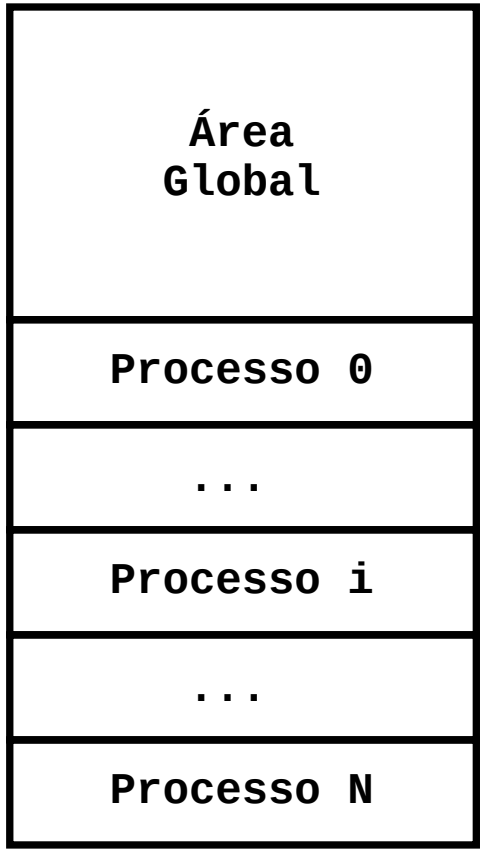
- Cada processo possui uma área local e todos partilham uma área global
- A partilha de tarefas é conseguida pela sincronização dos estados dos processos em diferentes partes da computação
- Essa sincronização corresponde na prática à cópia de segmentos de memória de um processo para outro



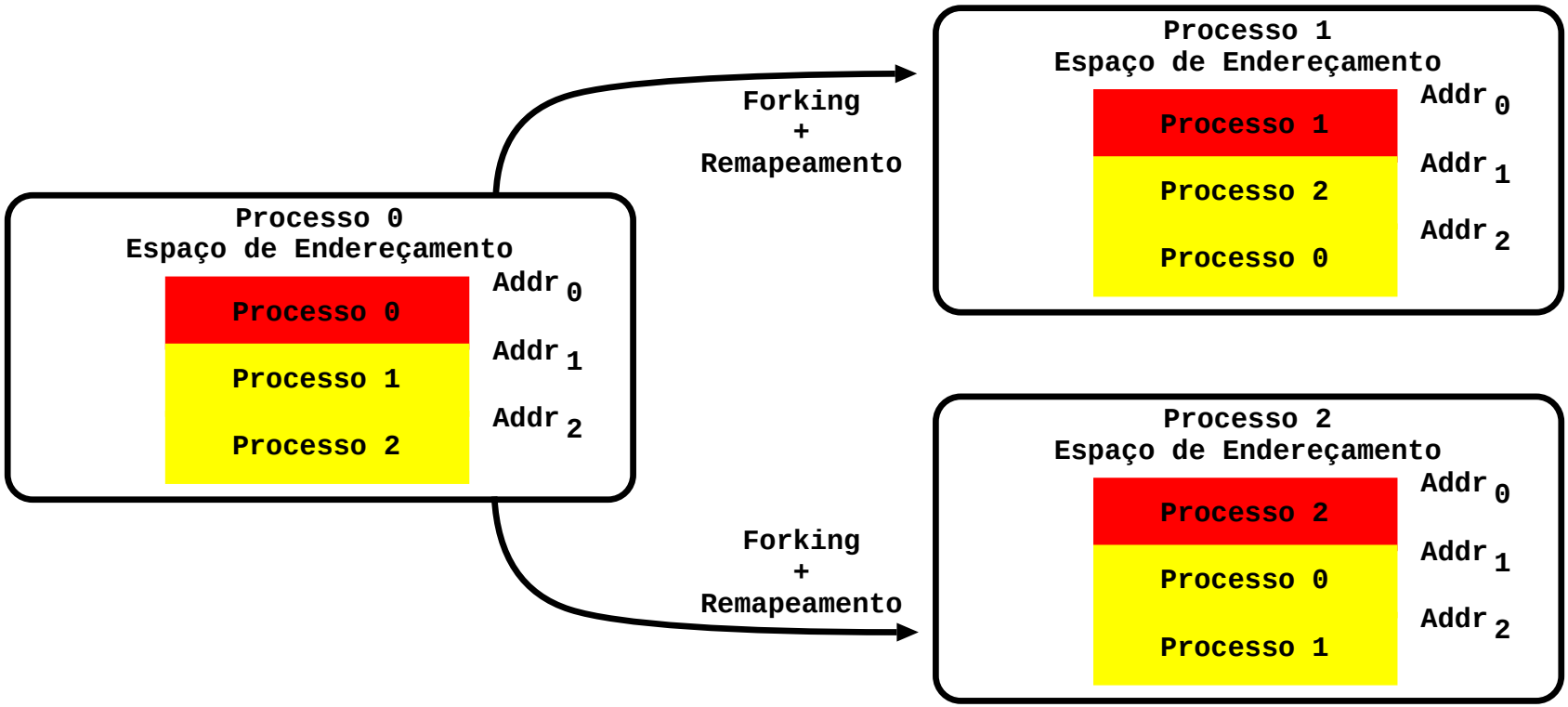
Técnicas Avançadas de Mapeamento de Memória

Problema: a cópia de segmentos de memória entre processos requer realocação de endereços de modo a fazerem sentido no novo espaço de endereçamento.

Solução: mapear a memória de tal modo que todos os processos vejam as suas próprias áreas no mesmo endereço, ou seja, **o espaço de endereçamento de cada processo**, dum ponto de vista individual, **tem início no mesmo endereço**.



Técnicas Avançadas de Mapeamento de Memória



Este esquema permite operações de cópia bastante eficientes, pois **evita a realocação de endereços**. Suponha, por exemplo, que o processo 2 quer copiar para o processo 1 um segmento de memória que começa no endereço `Addr` (do ponto de vista do processo 2). Então o endereço de destino deve ser $\texttt{Addr} + (\texttt{Addr}_2 - \texttt{Addr}_0)$.

Técnicas Avançadas de Mapeamento de Memória

```
map_addr = mmap(NULL, global_size + n_procs * local_size,
                PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
for (i = 0; i < n_procs; i++)
  proc(i) = map_addr + global_size + local_size * i;
for (p = 1; p < n_procs; p++)
  if (fork() == 0) {
    // unmap local regions
    remap_addr = map_addr + global_size;
    munmap(remap_addr, local_size * n_procs);
    // remap local regions
    for (i = 0; i < n_procs; i++) {
      proc(i) = remap_addr + local_size * ((n_procs + i - p) % n_procs);
      mmap(proc(i), local_size, PROT_READ | PROT_WRITE,
           MAP_SHARED | MAP_FIXED, fd, global_size + local_size * i);
    }
    break;
  }
```

A cópia de memória do processo 2 para o processo 1 a partir de `Addr` teria como endereço de destino `Addr + (proc(1) - proc(2))`.

Sincronização em Memória Partilhada

No **Parallel Rank Sort** os processos são independentes e não precisam de sincronizar no acesso à memória partilhada. Contudo, quando os processos atualizam estruturas de dados partilhadas (**zona crítica de código**) é necessário usar mecanismos que garantam **exclusão mútua**, i.e., que dois processos nunca estão simultaneamente dentro da mesma zona crítica.

Para além de garantir exclusão mútua, uma boa e correta solução para o **problema da zona crítica** deve verificar ainda as seguintes condições:

- Nenhum processo fora duma zona crítica deve bloquear outros
- Nenhum processo deve ter de esperar indefinidamente para entrar numa zona crítica
- A velocidade ou o número de CPU's disponíveis não deve ser relevante

Vamos ver dois mecanismos de sincronização:

- **Spinlocks** – de espera ativa
- **Semáforos** – sem espera ativa

Instruções Atómicas

Uma forma de garantir exclusão mútua de forma eficiente é proteger as zonas críticas de código através do uso de **instruções atómicas**:

- **Test and Set Lock (TSL)** – modifica o conteúdo de uma posição de memória para um valor pré-determinado e devolve o valor anterior
- **Compare And Swap (CAS)** – testa e troca o conteúdo de uma posição de memória em função de um valor esperado

A implementação deste tipo de instruções atómicas **necessita da ajuda do hardware**. Atualmente, as arquiteturas modernas implementam instruções atómicas do tipo TSL/CAS ou variantes.

Instruções Atómicas

```
// test and set lock
boolean TSL(boolean *target) {
  boolean aux = *target;
  *target = TRUE;
  return aux;
}

// compare and swap
boolean CAS(int *target, int expected, int new) {
  if (*target != expected)
    return FALSE;
  *target = new;
  return TRUE;
}
```

A execução das funções `TSL()` e `CAS()` têm de ser indivisível, i.e., nenhum outro processo pode aceder à posição de memória referenciada por `target` antes que a função complete a sua execução.

Exclusão Mútua com TSL

Questão: como usar a instrução TSL de forma a garantir exclusão mútua no acesso a uma zona crítica?

Solução: associar uma variável partilhada (**mutex lock**) à zona crítica e executar repetidamente a instrução TSL sobre essa variável até retornar o valor `FALSE`. O processo apenas acede à zona crítica quando a operação retorna `FALSE`, o que garante exclusão mútua.

```
#define INIT_LOCK(M)     M = FALSE
#define ACQUIRE_LOCK(M) while (TSL(&M))
#define RELEASE_LOCK(M) M = FALSE

INIT_LOCK(mutex);
... // non-critical section
ACQUIRE_LOCK(mutex);
... // critical section
RELEASE_LOCK(mutex);
... // non-critical section
```

Exclusão Mútua com CAS

```
#define INIT_LOCK(M)     M = 0
#define ACQUIRE_LOCK(M) while (!CAS(&M, 0, 1))
#define RELEASE_LOCK(M) M = 0

INIT_LOCK(mutex);
... // non-critical section
ACQUIRE_LOCK(mutex);
... // critical section
RELEASE_LOCK(mutex);
... // non-critical section
```

Spinlocks

Quando a solução para implementar exclusão mútua requer **espera ativa**, o mutex lock é chamado de **spinlock**.

Espera ativa pode ser um problema porque:

- Desperdiça tempo de CPU que outro processo poderia estar a utilizar
- Se o processo na posse do lock for interrompido (mudança de contexto) então nenhum outro processo consegue aceder ao lock e portanto será inútil dar tempo de CPU a esses processos
- Não satisfaz a condição de que nenhum processo deve ter de esperar indefinidamente para entrar numa zona crítica

Por outro lado, quando o **tempo de locking é muito curto**, espera-se que seja **mais vantajoso do que fazer troca de contexto**:

- Frequente em sistemas multiprocessador/multicore em que um processo tem a posse do lock e os restantes ficam em espera ativa

Spinlocks em Linux (include/linux/spinlock.h)

Iniciar o spinlock:

```
spin_lock_init(spinlock_t *spinlock)
```

Espera ativa até conseguir obter o spinlock:

```
spin_lock(spinlock_t *spinlock)
```

Tenta obter o spinlock mas não fica em espera caso não seja possível:

```
spin_trylock(spinlock_t *spinlock)
```

Libertar o spinlock:

```
spin_unlock(spinlock_t *spinlock)
```

Read-Write Spinlocks

Por vezes, a necessidade de garantir exclusão mútua no acesso a zonas críticas está apenas (ou maioritariamente) associada a operações de leitura sobre os dados partilhados.

Operações de leitura não exclusivas nunca levam a inconsistência dos dados, apenas as operações de escrita provocam esse problema.

Read-write spinlocks são uma solução alternativa que permitem **múltiplas operações de leitura mas apenas uma operação de escrita sobre zonas críticas**.

Read-Write Spinlocks em Linux (include/linux/rwlock.h)

Iniciar o spinlock:

```
rwlock_init(rwlock_t *rwlock)
```

Espera ativa até não existir qualquer outra operação de escrita:

```
read_lock(rwlock_t *rwlock)
```

Espera ativa até não existir qualquer outra operação de leitura ou escrita:

```
write_lock(rwlock_t *rwlock)
```

Read-Write Spinlocks em Linux (include/linux/rwlock.h)

Tenta obter o spinlock mas não fica em espera caso não seja possível:

```
read_trylock(rwlock_t *rwlock)
write_trylock(rwlock_t *rwlock)
```

Libertar o spinlock:

```
read_unlock(rwlock_t *rwlock)
write_unlock(rwlock_t *rwlock)
```

Spinlocks

Vantagens e inconvenientes:

- (+) É simples e fácil de verificar
- (+) Pode ser usado por um número arbitrário de processos
- (+) Suporta zonas críticas múltiplas
- (–) Com maior número de processos, espera ativa pode ser um problema
- (–) Quando temos múltiplas zonas críticas, é possível que os processos entrem em situações de **deadlock**

Semáforos

Foram introduzidos por Dijkstra em 1965 e permitem sincronizar o acesso a **recursos definidos por um número finito de instâncias**.

Um semáforo pode ser visto como que um **inteiro não negativo** que representa o número de instâncias disponíveis do recurso respetivo:

- Não é possível ler ou escrever o valor do semáforo diretamente, exceto para atribuir o seu valor inicial
- Não pode ser negativo pois quando este atinge o valor 0 (o que significa que todas as instâncias estão em utilização), os processos que tencionem usar o recurso ficam bloqueados até o semáforo voltar a ser não negativo

Existem dois tipos de semáforos:

- **Semáforos de Contagem** – que podem tomar qualquer valor
- **Semáforos Binários** – que apenas tomam os valores 0 e 1 (como os mutex locks)

Operações sobre Semáforos

Os semáforos podem ser acedidos através de **duas operações atómicas**:

- **DOWN** (ou **SLEEP** ou **WAIT**) – espera que o semáforo fique positivo e em seguida decrementa-o em uma unidade
- **UP** (ou **WAKEUP** ou **POST** ou **SIGNAL**) – incrementa o semáforo em uma unidade

```
down(semaphore S) {
  if (S == 0)
    suspend(); // suspend current process
  S--;
}

up(semaphore S) {
  S++;
  if (S == 1)
    wakeup(); // wakeup one waiting process
}
```

Implementação de Semáforos

A implementação deve garantir que nunca são executadas duas operações simultâneas de **DOWN** e/ou **UP** sobre o mesmo semáforo:

- Operações simultâneas de **DOWN** não podem decrementar o semáforo abaixo de zero
- Não se pode perder um incremento por **UP** se um **DOWN** acontecer simultaneamente

A implementação de semáforos tem por base mecanismos de sincronização que tentam **minimizar o tempo de espera ativa**. Existem duas abordagens para minimizar o tempo de espera ativa:

- Para uniprocessadores via o desativar de interrupções
- Para multiprocessadores/multicores via o desativar de interrupções mais o uso de instruções atómicas

Implementação de Semáforos em Uniprocessadores

```
typedef struct { // semaphore data structure
  int value;     // semaphore value
  PCB *queue;    // associated queue of waiting processes
} semaphore;

init_semaphore(semaphore S) {
  S.value = 1;
  S.queue = EMPTY;
}

down(semaphore S) {
  disable_interrupts();
  if (S.value == 0) { // avoid busy waiting
    add_to_queue(current_PCB, S.queue);
    suspend();
    // kernel reenables interrupts just before restarting here
  } else {
    S.value--;
    enable_interrupts();
  }
}
```

Implementação de Semáforos em Uniprocessadores

```
up(semaphore S) {
  disable_interrupts();
  if (S.queue != EMPTY) {
    // keep semaphore value and wakeup one waiting process
    waiting_PCB = remove_from_queue(S.queue);
    add_to_queue(waiting_PCB, OS_ready_queue);
  } else {
    S.value++;
  }
  enable_interrupts();
}
```

Implementação de Semáforos em Multiprocessadores

```
typedef struct { // semaphore data structure
  boolean mutex; // to guarantee atomicity
  int value;     // semaphore value
  PCB *queue;    // associated queue of waiting processes
} semaphore;

init_semaphore(semaphore S) {
  INIT_LOCK(S.mutex);
  S.value = 1;
  S.queue = EMPTY;
}
```

Implementação de Semáforos em Multiprocessadores

```
down(semaphore S) {
  disable_interrupts();
  ACQUIRE_LOCK(S.mutex); // short busy waiting time
  if (S.value == 0) {
    add_to_queue(current_PCB, S.queue);
    RELEASE_LOCK(S.mutex);
    suspend();
    // kernel reenables interrupts just before restarting here
  } else {
    S.value--;
    RELEASE_LOCK(S.mutex);
    enable_interrupts();
  }
}
```

Implementação de Semáforos em Multiprocessadores

```
up(semaphore S) {
  disable_interrupts();
  ACQUIRE_LOCK(S.mutex); // short busy waiting time
  if (S.queue != EMPTY) {
    // keep semaphore value and wakeup one waiting process
    waiting_PCB = remove_from_queue(S.queue);
    add_to_queue(waiting_PCB, OS_ready_queue);
  } else {
    S.value++;
  }
  RELEASE_LOCK(S.mutex);
  enable_interrupts();
}
```

Semáforos POSIX

Os semáforos POSIX estão disponíveis em duas versões:

- **Named Semaphores** – são acedidos por nome e por isso podem ser utilizados por todos os processos que conheçam esse nome
- **Unnamed Semaphores** – apenas existem em memória e por isso só podem ser utilizados pelos processos que partilham o mesmo espaço de memória

Ambas as versões funcionam de igual modo, apenas diferem na forma como os semáforos são iniciados e libertados.

Criar Semáforo com Nome

```
sem_t *sem_open(char *name, int oflag)
sem_t *sem_open(char *name, int oflag, mode_t mode, int value)
```

`sem_open()` cria um novo semáforo com nome ou abre um já existente e retorna o endereço do semáforo. Em caso de erro retorna `SEM_FAILED`.

- `name` é o nome que identifica o semáforo (por convenção, é comum o primeiro caractere do nome ser '/' e o nome não conter mais '/'s)
- `oflag` permite especificar opções de criação/abertura: `O_CREAT` indica que um novo semáforo deve ser criado (caso ainda não exista); `O_EXCL` indica que o semáforo deve ser exclusivo (falha caso já tenha sido criado); `0` indica que se pretende abrir um semáforo já existente
- `mode` especifica as opções de acesso (apenas importante quando se cria um novo semáforo com a opção `O_CREAT`)
- `value` especifica o valor inicial do semáforo (apenas importante quando se cria um novo semáforo com a opção `O_CREAT`)

Fechar Semáforo com Nome

```
int *sem_close(sem_t *sem)
```

`sem_close()` fecha o acesso a um semáforo com nome e liberta todos os recursos do processo associados a esse semáforo (o valor do semáforo não é afetado). Retorna 0 se OK, -1 se erro.

- `sem` é o endereço que identifica o semáforo a fechar

Por omissão, os recursos associados a semáforos abertos por um processo são libertados quando o processo termina a sua execução (semelhante ao que acontece com ficheiros abertos no contexto de um processo).

Remover Semáforo com Nome

```
int *sem_unlink(char *name)
```

`sem_unlink()` remove o nome do semáforo do sistema (i.e., deixa de ser possível abrir o semáforo com `sem_open()`) e, se não existirem referências por fechar para o semáforo, o semáforo é também destruído. Caso contrário, o semáforo é apenas destruído quando não existirem referências por fechar para o semáforo. Retorna 0 se OK, -1 se erro.

- `name` é o nome que identifica o semáforo a remover

Criar Semáforo sem Nome

```
int sem_init(sem_t *sem, int pshared, int value)
```

`sem_init()` cria um semáforo sem nome para ser partilhado entre threads ou entre processos. Retorna 0 se OK, -1 se erro.

- `sem` é o endereço que identifica o semáforo sem nome
- `pshared` especifica se o semáforo é para ser partilhado entre threads (`0`) ou entre processos (`1`)
- `value` especifica o valor inicial do semáforo

Para partilhar o semáforo entre threads, este deve estar localizado num **endereço de memória visível por todos os threads**.

Para partilhar o semáforo entre processos, este deve estar localizado numa **região de memória partilhada por todos os processos**.

Remover Semáforo sem Nome

```
int sem_destroy(sem_t *sem)
```

`sem_destroy()` destrói um semáforo sem nome. Retorna 0 se OK, -1 se erro.

- `sem` é o endereço que identifica o semáforo a destruir

Destruir um semáforo que outros threads ou processos possam ainda utilizar leva a um comportamento desconhecido, a menos que entretanto o semáforo seja novamente criado por chamada da função `sem_init()`.

Operações sobre Semáforos com/sem Nome

```
int sem_post(sem_t *sem)
int sem_wait(sem_t *sem)
int sem_trywait(sem_t *sem)
```

`sem_post()` incrementa o valor do semáforo enquanto que `sem_wait()` e `sem_trywait()` decrementam o valor do semáforo. `sem_wait()` bloqueia enquanto o semáforo estiver a 0, enquanto que `sem_trywait()` evita o bloqueio retornando um valor de erro em lugar de bloquear. Todos retornam 0 se OK, -1 se erro.

- `sem` é o endereço que identifica o semáforo a incrementar ou decrementar

Sequência de Utilização Básica Semáforo com Nome

```c
#define SEM_NAME "/mysem"

int main() {
  sem_t *sem;
  sem = sem_open(SEM_NAME, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 1);
  ... // use sem_wait()/sem_post() to increment/decrement semaphore
  sem_close(sem); // close semaphore
  sem_unlink(SEM_NAME); // destroy semaphore name
}
```

Sequência de Utilização Básica Semáforo sem Nome

```c
sem_t sem; // unnamed semaphore to use with threads

int main() {
  sem_init(&sem, 0, 1); // create semaphore
  ... // use sem_wait()/sem_post() to increment/decrement semaphore
  sem_destroy(&sem); // destroy semaphore
}
```

```c
sem_t *sem; // unnamed semaphore to use with processes

int main() {
  sem = (sem_t *) shmget(...); // allocate shared memory for semaphore
  sem_init(sem, 1, 1); // create semaphore
  ... // use sem_wait()/sem_post() to increment/decrement semaphore
  sem_destroy(sem); // destroy semaphore
}
```

O Problema do Barbeiro Dorminhoco

O problema do barbeiro dorminhoco é um problema clássico de IPC:

- Uma barbearia tem um dado número de barbeiros e um dado número de cadeiras (`NCHAIRS`) para os clientes em espera
- Sempre que um barbeiro não tem clientes para atender ele aproveita para dormir um pouco
- Quando um cliente chega à barbearia, este tem de acordar um barbeiro para lhe cortar o cabelo
- Se entretanto chegarem mais clientes e todos os barbeiros estiverem ocupados estes devem esperar sentados (se existirem cadeiras livres) ou então deixar a barbearia (se todas as cadeiras estiverem ocupadas)

O Problema do Barbeiro Dorminhoco

```
int waiting = 0; semaphore clients = 0, barbers = 0, mutex = 1;

client() {
  down(mutex);
  if (waiting >= NCHAIRS) { up(mutex); exit(1); }
  waiting++;
  up(clients); // wakeup a barber if necessary
  up(mutex);
  down(barbers); // wait if there are no barbers available
  get_hair_cut();
}

barber() {
  while(1) {
    down(clients); // sleep if there are no clients
    down(mutex);
    waiting--;
    up(barbers); // ready to cut hair
    up(mutex);
    cut_hair();
  }
}
```