Programação Paralela

Parte II

Programação em Memória Partilhada

Ricardo Rocha
ricroc@ncc.up.pt
DCC-FCUP

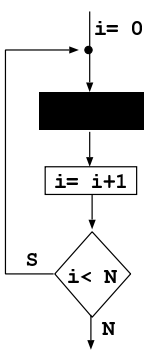
MI/MIAC 2002/03

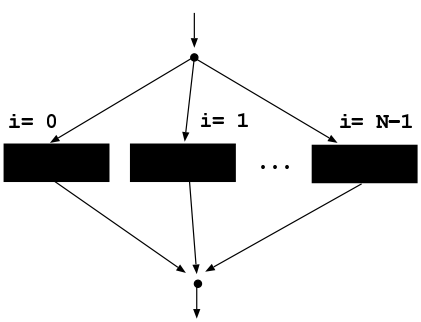
Paralelismo nos Dados

"Realizar a mesma operação sobre cada componente dos dados."

Paralelismo nos dados é uma das técnicas mais simples e poderosas de explorar paralelismo.

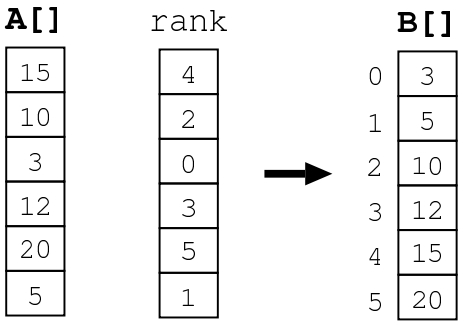
- Frequente em problemas científicos e de engenharia.
- Os dados surgem normalmente sob a forma de matrizes multidimensionais.
- Os ciclos são os principais candidatos a serem paralelizados.





Exemplo: Rank Sort

- Seja A_N um vector com N elementos. A ideia é construir um vector ordenado B_N a partir dos elementos de A_N. Para isso basta, para cada elemento de A_N, determinar a sua posição (*rank*) relativa no vector.
- A posição de um dado elemento $A[k]$, define-se como sendo o número de elementos em $A[\,]$ que são menores que $A[k]$.



- O algoritmo é facilmente paralelizável, pois a posição de cada elemento na lista pode ser calculada de forma independente.

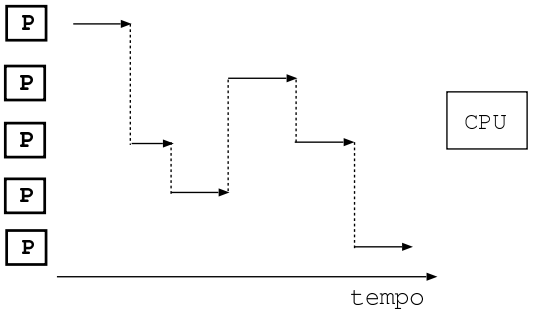
Rank Sort (rank.c)

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

main () {
  ...
  for (i = 0; i < N; i++)  // paralelizar aqui
    calcula_posicao(i);
  ...
}

calcula_posicao(int i) {
  int t, j, rank;

  t = A[i]; rank = 0; j = i;
  do {
    j = (j + 1) % N;  // i+1, ..., N-1, 0, 1, ..., i-1, i
    if (t > A[j]) rank++;
  } while (j != i);
  B[rank] = t;  // coloca valor de A[] na posição correcta de B[]
}
```

- Para paralelizar o algoritmo é necessário poder criar diferentes instâncias do programa, cada uma dedicada a uma diferente tarefa.

Processos em Unix

- Um processo é a abstracção de um programa em execução. Um programa pode ter várias instâncias em execução.
- Em máquinas uniprocessador, em cada instante, apenas um processo está em execução. No entanto, em pequenas fracções de tempo, vários processos podem ser executados, dando ao utilizador a ilusão de paralelismo.



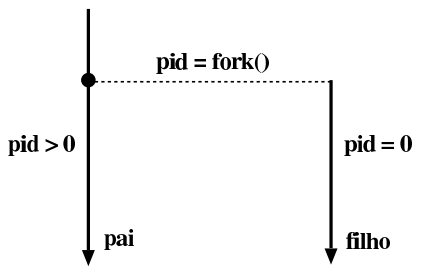
- O sistema operativo mantêm uma tabela de processos onde guarda toda a informação relativa ao estado de cada processo quando suspenso. Quando um processo é novamente escalonado para execução essa informação é recuperada tal como se nada tivesse acontecido.

Criação de Processos em Unix

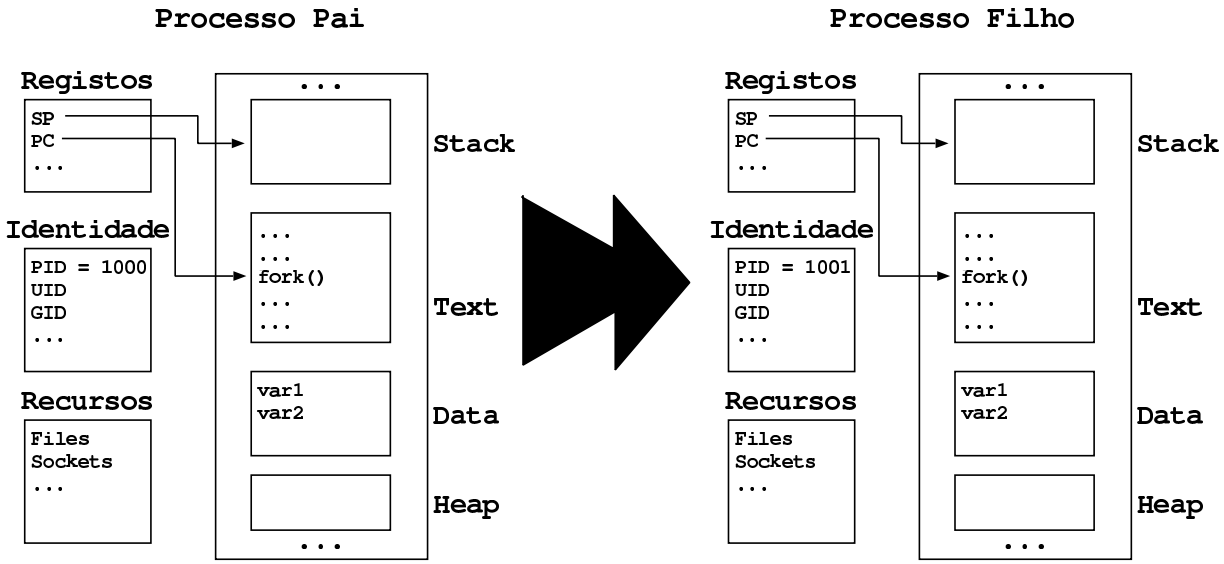
```
pid_t fork(void);
```

- fork() retorna 0 ao processo filho e retorna o PID do filho ao processo pai. Caso não seja possível criar um novo filho retorna -1.
- Como distinguir a execução do processo filho da do processo pai?

```
pid_t pid;
...
pid = fork();
if (pid == 0) {
  <código processo filho>
} else {
  <código processo pai>
}
<código possivelmente comum>
```



Criação de Processos em Unix



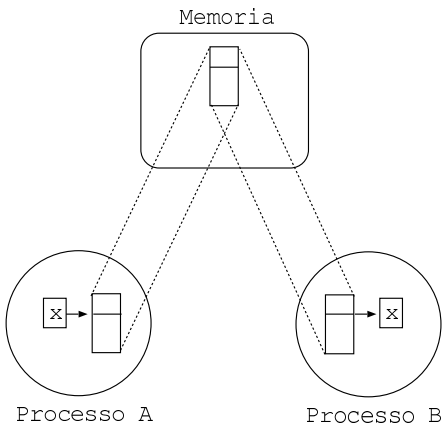
Parallel Rank Sort (rank_par.c)

```
main () {
  ...
  for (i = 0; i < N; i++)
    if (fork() == 0) {
      calcula_posicao(i);  // cada processo executa uma tarefa
      exit(0);
    }
  for (i = 0; i < N; i++)
    wait(&status);  // pai espera que todos os filhos terminem
  for (i = 0; i < N; i++)
    printf("%d\n", B[i]);  // o que escreve aqui?
  ...
}
```

- Conseguimos lançar N processos, cada um executa `calcula_posicao()` para um valor diferente de `i`. O que é que o programa escreve no final?
- Cada processo filho herda uma cópia das variáveis do processo pai, mas ao alterar essas variáveis o pai não vê as alterações. As alterações feitas sobre `B[]` não ficaram visíveis e o pai escreve uma sequência de zeros!
- **Solução:** `B[]` tem de ser partilhado.

Memória Partilhada em Unix

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



- O segmento partilhado é conhecido de ambos os processos. 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() retorna o inteiro que identifica o segmento. Caso não seja possível alocar o segmento retorna -1.
- key é a chave que identifica o segmento pretendido. Outros processos podem aceder ao mesmo segmento se especificarem a mesma chave. A constante `IPC_PRIVATE` garante que um novo segmento é sempre 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 exacto).
- 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 num endereço de memória do 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 (arredondado para um múltiplo das páginas do sistema). Se `NULL`, o sistema escolhe um 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 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 de memória mapeado.

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 fim de um processo liberta automaticamente o mapeamento mas não remove o segmento. shmctl() deve ser explicitamente invocado por um dos processos.

O número de segmentos de memória partilhada disponíveis é limitado.

- ipcs para ver os segmentos que estão a ser usados.
- ipcrm para remover um segmento.

Sequência de Utilização Básica

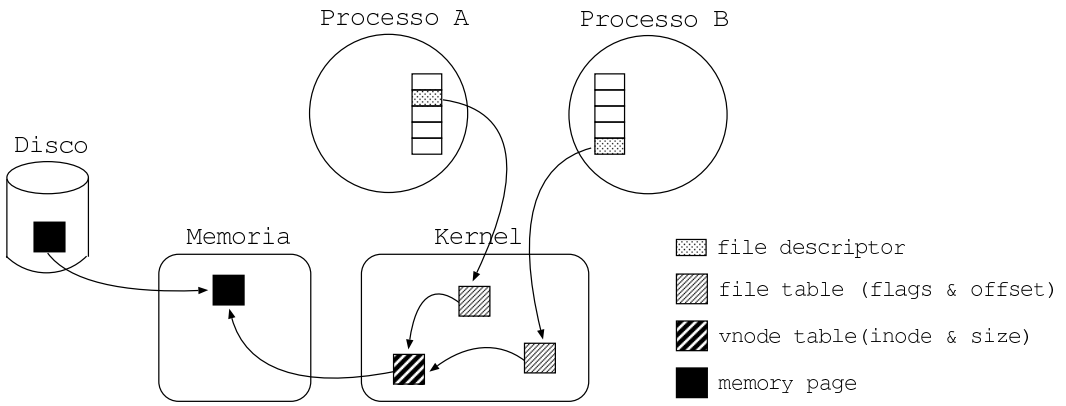
```
int segment_id, segment_size;
char *shared_memory;
...
segment_size =  getpagesize();
segment_id = shmget(IPC_PRIVATE, segment_size, S_IRUSR | S_IWUSR);
shared_memory = (char *) shmat(segment_id, NULL, 0);
...
sprintf(shared_memory, "Hello World.");
...
shmdt(shared_memory);
shmctl(segment_id, IPC_RMID,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


Parallel Rank Sort com Memória Partilhada (rank_shm.c)

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

main () {
  int shmid;
  ...
  // alocar e mapear um segmento para representar o vector B[]
  shmid = shmget(IPC_PRIVATE, N * sizeof(int), S_IRUSR | S_IWUSR);
  B = (int *) shmat(shmid, NULL, 0);
  for (i = 0; i < N; i++)
    if (fork() == 0) {
      calcula_posicao(i);
      exit(0);
    }
  for (i = 0; i < N; i++)
    wait(&status);
  for (i = 0; i < N; i++)
    printf("%d\n", B[i]);
  // libertar e remover o segmento
  shmdt(B);
  shmctl(shmid, IPC_RMID, NULL);
}
```

Mapeamento de Ficheiros em Memória

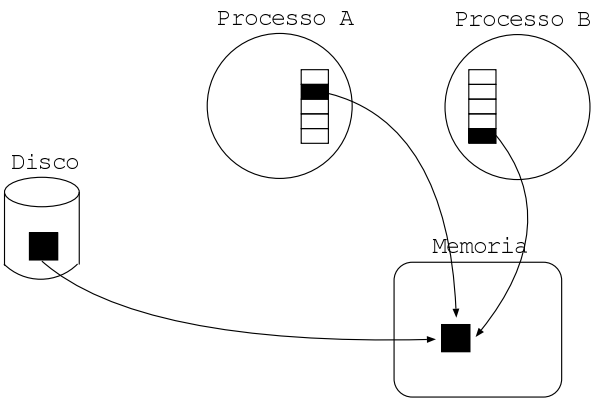
- A comunicação entre processos por memória partilhada também pode ser conseguida via ficheiros partilhados.
- O acesso a ficheiros em Unix é habitualmente conseguido por uso das funções `open()`, `read()`, `write()`, `lseek()`, e `close()`.



- Dois processos que acedam a um mesmo ficheiro, sincronizam na estrutura de dados *vnode* associada ao ficheiro. A atomicidade na escrita e leitura do ficheiro é garantida pelas operações de `read()` e `write()`.

Mapeamento de Ficheiros em Memória

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



- O 1^o processo a aceder à página origina um *page-fault*; o kernel copia a página do disco para memória; quando o 2^o processo aceder à página, o kernel limita-se a adicionar na tabela de páginas do processo um apontador para a página em memória.

Mapeamento de Ficheiros em Memória

Sequência de utilização para mapear um ficheiro em disco 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

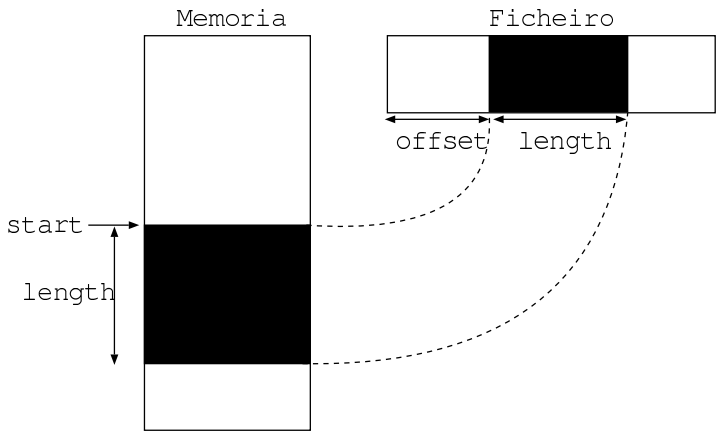
```
void *mmap(void *start, size_t length, int prot, int flags,
                  int fd, off_t offset);
```

- mmap() retorna o endereço de memória que inicia a região 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). Se NULL, o sistema escolhe um 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.
- 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 reflectem no conteúdo ficheiro (normalmente utilizada para *debugging*).

Mapear uma Região de um Ficheiro em Memória

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

- 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).



Libertar uma Região de Memória Mapeada

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

- munmap() liberta o mapeamento feito e a região deixa de ficar associada a um endereço de memória. Retorna 0 se OK, -1 se erro.
- start é o endereço inicial de memória da região a libertar.
- length é o tamanho a libertar.

Sequência de Utilização Básica

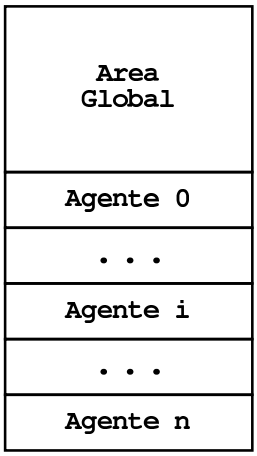
```
int fd, region_size;
void *mapped_mem;
...
region_size =  getpagesize();
fd = open("mapfile", O_RDWR | O_CREAT | O_TRUNC,  S_IRUSR | S_IWUSR);
lseek(fd, region_size, SEEK_SET);
write(fd, "", 1);
mapped_mem = mmap(NULL, region_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
...
sprintf(mapped_mem,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


Parallel Rank Sort com Memória Mapeada (rank_mmap.c)

```
int A[N], *B;
#define MMAP_SIZE N * sizeof(int)

main () {
  int fd;
  ...
  // mapear um ficheiro em memória para representar o vector B[]
  fd = open("mapfile", O_RDWR | O_CREAT | O_TRUNC,  S_IRUSR | S_IWUSR);
  lseek(fd, MMAP_SIZE, SEEK_SET);
  write(fd, "", 1);
  B = (int *) mmap(NULL, MMAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  for (i = 0; i < N; i++)
    if (fork() == 0) {
      calcula_posicao(i);
      exit(0);
    }
  for (i = 0; i < N; i++)
    wait(&status);
  for (i = 0; i < N; i++)
    printf("%d\n", B[i]);
  // libertar a região de memória mapeada
  munmap(B, MMAP_SIZE);
}
```

Técnicas Avançadas de Mapeamento de Memória

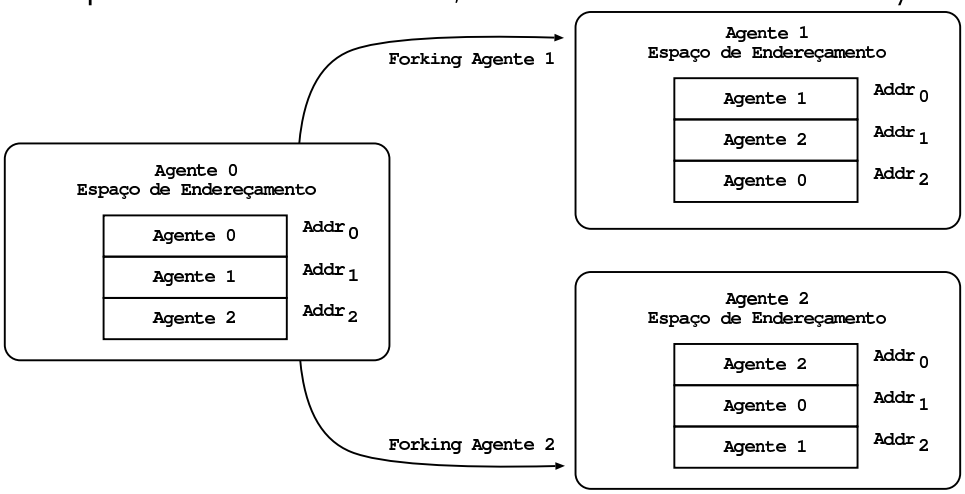
- Considere um conjunto de agentes tal como na figura que se segue:



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

Técnicas Avançadas de Mapeamento de Memória

- **Problema:** a cópia de segmentos de memória entre agentes requer realocação de endereços de modo a fazerem sentido no novo espaço de endereçamento.
- **Solução:** mapear a memória de tal modo que todos os agentes vejam as suas próprias áreas no mesmo endereço, ou seja, o espaço de endereçamento de cada agente, dum ponto de vista individual, tem início no mesmo endereço.



Técnicas Avançadas de Mapeamento de Memória

```
mapped_mem = mmap(NULL, global_space + n_agentes * local_space,
                  PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
for (i = 0; i < n_agentes; i++)
  agente(i) = mapped_mem + global_space + local_space * i;
for (w = 1; w < n_agentes; w++)
  if (fork() == 0) {
    // libertar as regiões locais
    remap_addr = mapped_mem + global_space;
    munmap(remap_addr, local_space * n_agentes);
    // remapear as regiões locais
    for (i = 0; i < n_agentes; i++) {
      agente(i) = remap_addr + local_space * ((n_agentes + i - w) % n_agentes);
      mmap(agente(i), local_space, PROT_READ | PROT_WRITE,
           MAP_SHARED | MAP_FIXED, fd, global_space + local_space * i);
    }
  }
```

- Este esquema permite operações de cópia bastante eficientes, pois evita a realocação de endereços.
- Suponha que o agente 2 quer copiar para o agente 1 um segmento de memória que começa no endereço `Addr` (do ponto de vista do agente 2). Então o endereço de destino deve ser `Addr + (agente(1) - agente(2))`.

Sincronização em Memória Partilhada

- No exemplo do *Parallel Rank Sort* os processos são completamente independentes e não precisam de sincronizar o acesso à memória partilhada (vector B[ ]).
- Contudo, se vários processos actualizarem estruturas de dados partilhadas (**zona crítica de código**), é necessário usar mecanismos que garantam exclusão mútua.
- Uma boa solução deve verificar as seguintes condições:
  - Dois processos não podem estar simultaneamente dentro da mesma zona crítica.
  - Nenhum processo a executar fora de uma zona crítica pode 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 activa.
  - **Semáforos** – sem espera activa.

A Instrução *Test and Set Lock* (TSL)

- A instrução TSL testa e modifica o conteúdo de uma posição de memória de modo atómico.
- A instrução TSL corresponde à função:

```
int TSL(int *m) {
  int r;
  r = *m;
  *m = 1;
  return r;
}
```

- A execução da função `TSL()` tem de ser indivisível, isto é, nenhum outro processo pode aceder à posição de memória `*m` antes que a função seja completamente executada.
- A instrução TSL para ser implementada necessita de uma pequena ajuda do hardware. Actualmente, quase todas as arquitecturas implementam uma instrução do tipo TSL ou variantes.

Exclusão Mútua com TSL

- **Questão:** Como usar a instrução `TSL()` de forma a garantir exclusão mútua no acesso a uma zona crítica?
- **Solução:** Utilizar uma variável partilhada (**lock**) para sincronizar o acesso a zonas críticas, obrigando os processos a activar o *lock* antes de entrarem na zona crítica.

```
...             // zona não crítica
LOCK(lock);     // espera activa até conseguir activar o lock
...             // zona crítica
UNLOCK(lock);   // liberta o lock
...             // zona não crítica
```

- Um processo apenas acede à zona crítica quando a operação `LOCK(lock)` for bem sucedida.

Uma Implementação para Arquitecturas x86

```
typedef volatile int lockvar;

#define swap(reg, adr)                          \
({                                              \
   char _ret;                                   \
   asm volatile ("xchgb %0,%1"                  \
        : "=q" (_ret), "=m" (*(adr))            \
        : "m"  (*(adr)), "0"  (reg));           \
   _ret;                                        \
})
#define TSL(lock)       swap(1, lock);
#define INITLOCK(lock)  lock = 0;
#define LOCK(lock)      while (TSL(&(lock)) != 0);
#define UNLOCK(lock)    lock = 0;

...
lockvar var;
INIT_LOCK(var);
...
LOCK(var);
...
UNLOCK(var);
...
```


Vantagens e Inconvenientes dos Locks

- **(+)** 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 activa pode ser um problema.
- **(–)** É possível os processos entrarem em *deadlock*.

Semáforos

- Os semáforos foram introduzidos em 1965 por Dijkstra e permitem sincronizar processos no acesso a zonas críticas sem envolver espera activa.
- Um semáforo é definido como um inteiro não negativo, ao qual estão associadas duas operações atómicas (indivisíveis): DOWN (ou SLEEP) e UP (ou WAKEUP).
- As operações DOWN e UP correspondem às funções:

```
DOWN(s) {
  if (s == 0)
    suspende execução do processo;
  s--;
}

UP(s) {
  s++;
  if (s == 1)
    retoma um processo suspenso em s;
}
```

- *Mutexs* ou semáforos binários: apenas tomam os valores 0 e 1.

Alocar um Conjunto de Semáforos

```
int semget(key_t key, int nsems, int flags);
```

- semget() retorna o inteiro que identifica o conjunto de semáforos. Caso não seja possível criar o conjunto retorna -1.
- key é a chave que identifica o conjunto pretendido. Outros processos podem aceder ao mesmo conjunto se especificarem a mesma chave. A constante `IPC_PRIVATE` garante que um novo conjunto é sempre atribuído.
- nsems é o número de semáforos a alocar (se o conjunto já tiver sido alocado podemos especificar `nsems` como 0).
- flags permite especificar opções de alocação: `IPC_CREAT` indica que um novo conjunto deve ser criado (caso não exista); `IPC_EXCL` indica que o conjunto deve ser exclusivo (falha caso contrário); `S_IRUSR`, `S_IWUSR`, `S_IROTH` e `S_IWOTH` indicam as permissões de leitura e escrita.

Remover um Conjunto de Semáforos

```
int semctl(int semid, int semnum, int cmd, ...);
```

- semctl() remove um conjunto de semáforos (a remoção é imediata). Retorna 0 se OK, -1 se erro.
- semid é o inteiro que identifica o conjunto de semáforos.
- semnum pode ser 0.
- cmd deve ser `IPC_RMID`.

Operações sobre Semáforos

```
int semop(int semid, struct sembuf *semops, unsigned nsemops);
```

- semop() executa uma série de operações sobre um conjunto de semáforos. Retorna 0 se OK, -1 se erro.
- semid é o inteiro que identifica o conjunto de semáforos.
- semops vector de operações a executar sobre os semáforos do conjunto, em que cada operação é representada pela estrutura:

```
struct sembuf {
  unsigned short sem_num;  // número do semáforo
  short sem_op;            // operação a realizar
  short sem_flg;           // opções da operação
}
```

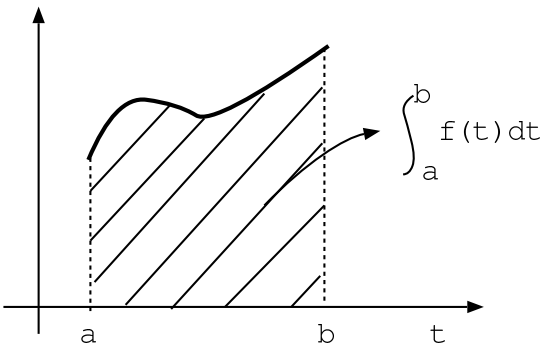
  Valores de `sem_op` < 0 implicam espera do processo e lock do semáforo (se resultado semáforo = 0). Valores de `sem_op` > 0 libertam o semáforo.
- nsemops tamanho do vector de operações.

Uma implementação de Semáforos em Unix

```
#define UP(semid, semnum)     {struct sembuf up = {semnum, 1, 0};      \
                               semop(semid, &up, 1); }
#define DOWN(semid, semnumn)  {struct sembuf down = {semnum, -1, 0};   \
                               semop(semid, &down, 1); }
#define INIT(semid, semnum)   UP(semid, semnum);

int init_sems(int nsems) {
  int semid, i;
  semid = semget(IPC_PRIVATE, nsems, S_IRUSR | S_IWUSR);
  for (i = 0; i < nsems; i ++)
    INIT(semid, i);
  return semid;
}

void free_sems(int semid) {
  semctl(semid, 0, IPC_RMID);
}
```


Exemplo: Integração Numérica

- O integral de uma função $f(t)$ no intervalo $[a, b]$ pode ser calculado por aproximação numérica, usando a regra dos trapézios.

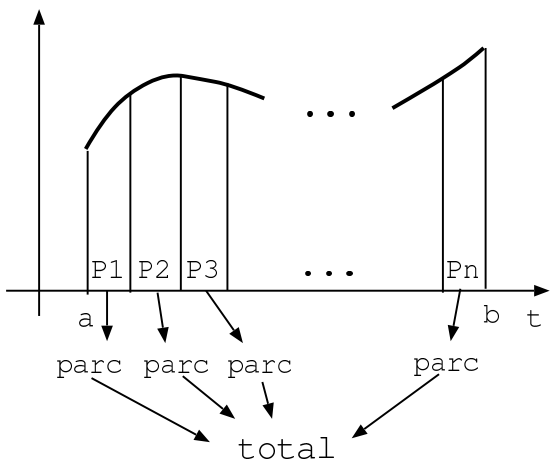


- Seja n o número de sub-intervalos e h o espaçamento de cada sub-intervalo, então:

$$area = h(\frac{f(a)}{2} + f(a+h) + f(a+2h) + \ldots + f(a+nh) + \frac{f(b)}{2})$$

Integração Numérica: Como Paralelizar?

- Dividir os sub-intervalos pelos processadores. Cada processador calcula uma área parcial que contribui para a área total.



- O que fazer se o número de sub-intervalos for maior que o número de processadores disponíveis?

Integração Numérica com Semáforos (integra.c)

```
#define NPROCS 5
#define A 0
#define B 10
#define N 20
#define H (((float)B - (float)A) / N)

float *soma;
int mutex;

void integral(int region) {
  float t, soma_parcial;
  int i;
  t = A + region * (B - A) / NPROCS;  // início do intervalo
  soma_parcial = 0;
  for (i = 0; i < N / NPROCS; i++) {
    soma_parcial += f(t);
    t += H;
  }
  DOWN(mutex, 1);
  *soma = *soma + soma_parcial;
  UP(mutex, 1);
}
```

Integração Numérica com Semáforos (integra.c)

```
main() {
  int i, shmid, status;

  // alocar e mapear um segmento de memória partilhada para a soma
  shmid = shmget(IPC_PRIVATE, sizeof(float), S_IRUSR | S_IWUSR);
  soma = (float *) shmat(shmid, NULL, 0);
  // alocar um mutex
  mutex = init_sems(1);
  *soma = 0;
  for (i = 0; i < NPROCS; i++)
    if (fork() == 0) {
      integral(i);
      exit(0);
    }
  for (i = 0; i < NPROCS; i++)
    wait(&status);
  printf("Area total= %f\n", H * (*soma + (f(B) - f(A)) / 2));
  // remover o mutex
  free_sems(mutex);
  // libertar e remover o segmento de memória partilhada
  shmdt(soma);
  shmctl(shmid, IPC_RMID, NULL);
}
```


Exemplo: O Problema do Barbeiro Dorminhoco

O problema do barbeiro dorminhoco é um dos problemas clássicos de IPC.

- Uma barbearia tem um dado número de barbeiros e um dado número de cadeiras (NCADEIRAS) 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 com Semáforos

```
int em_espera = 0;
semaphore clientes = 0, barbeiros = 0, mutex = 1;

void cliente(void) {
  DOWN(mutex);
  if (em_espera >= NCADEIRAS) { UP(mutex); exit(1); }
  em_espera = em_espera + 1;
  UP(clientes);      // acorda um barbeiro se necessário
  UP(mutex);
  DOWN(barbeiros);   // espera se não existirem barbeiros disponíveis
  obtem_corte();
}

void barbeiro(void) {
  while(1) {
    DOWN(clientes);  // dorme se não existirem clientes
    DOWN(mutex);
    UP(barbeiros);   // pronto para cortar o cabelo
    em_espera = em_espera - 1;
    UP(mutex);
    corta_cabelo();
  }
}
```


Bibliografia

- ***Advanced Linux Programming.*** M. Mitchell, J. Oldham, A. Samuel. New Riders. `http://www.advancedlinuxprogramming.com` **(Capítulos 3 e 5. Biblioteca ref. 344)**
- ***Advanced Programming in the UNIX Environment.*** W. R. Stevens. Addison Wesley. **(Págs 187–193, 407–413 e 457–473. Biblioteca ref. 1796)**