

# Sincronização e memória compartilhada no Linux



## Memória compartilhada

- Modelo de memória UNIX
  - Processo *aloca (allocate)* um segmento no qual deseja compartilhar
  - Processos que desejam acessar o segmento alocado, devem se *vincular (attach)* ao segmento
  - Após a utilização da memória compartilhada, os processos devem se *desvincular (detach)*
  - Quando a memória compartilhada não estiver mais em uso, a mesma deve ser *liberada (deallocate)*



## Alocação

- `shmget(key_t key, int size, int shmflg);`
  - Key: valor do identificador utilizado para designar a memória compartilhada. `IPC_PRIVATE`: utiliza um valor key único
  - Size: tamanho da memória, arredondado para cima de acordo com o tamanho da página
  - Shmflg: flags para controlar a criação da memória compartilhada
    - `IPC_CREAT`: cria um novo segmento quando, um valor key é passado
    - `IPC_EXCL`: exclusive, usado em conjunto com `IPC_CREAT`, caso o segmento já exista, retorna erro
    - `MODE FLAGS`: flags de acesso 9 bits (`TODOS/GRUPO/DONO`). Utilizar constantes predefinidas (ver man 2 stat). Ex: `S_IRUSR` `S_IWUSR` : leitura e escrita para o dono da memória compartilhada



## Vincular (attach)

- `shmat(int shmid, const void *shm_addr, int shmflg)`
  - shmid: identificador retornado durante a alocação
  - shm\_addr: endereço no espaço local do processo, se NULL o SO escolhe um endereço livre
  - shm\_flg
    - `SHM_RND`: arredonda para o próximo endereço múltiplo do tamanho de página
    - `SHM_RDONLY`: somente leitura



## Desvincular (detach) e Liberar (deallocate)

- Desvincular
  - int shmd(const void \*shmaddr);
- Liberar
  - shmctl(int shmid, int cmd, struct shmid\_ds \*bf);
- Verificar memórias compartilhadas
  - ipcs



## Exemplo

```
#define SHM_SIZE 4096
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main(){
    int shmid;
    char* saddr;
    // Allocate
    shmid= shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT| S_IRUSR | S_IWUSR);
    // Attach
    saddr= (char *)shmat(shmid, 0, 0);
    if (fork()==0){//children
        saddr[0]=65;
        sleep(5);
        printf("Filho %s", saddr);
        return 0;
    }
    saddr[1]=66;
    sleep(5);
    printf(" Pai %s" , saddr);
    return 0;
    wait();
    shmctl(shmid, IPC_RMID, 0);
}
```



## Semáforos

- Linux: duas implementações POSIX e SystemV
  - POSIX: utilizado em threads
  - SystemV: processos e threads. Complicado



## Alocação

- `int semget(key_t key, int nsems, int semflg);`
  - key: identificador do semáforo. `IPC_PRIVATE` cria um identificador único
  - nsems: número de semáforos no conjunto
  - Semflg
    - `IPC_CREAT`: cria um novo segmento quando, um valor key é passado
    - `IPC_EXCL`: exclusive, usado em conjunto com `IPC_CREAT`, caso o segmento já exista, retorna erro
    - `MODE FLAGS`: flags de acesso 9 bits (TODOS/GRUPO/DONO). Utilizar constantes predefinidas (ver man 2 stat). Ex: `S_IRUSR`  
`S_IWUSR` : leitura e escrita para o dono da memória compartilhada



## Inicialização

- **int semctl(int semid, int semnum, int cmd, union semun arg);**
  - semid= identificador retornado pelo semget
  - semnum= número do semáforo
  - Cmd
    - **GETALL**
    - **GETNCNT**
    - **GETPID**
    - **GETVAL**
    - **GETZCNT**
    - **SETALL**
    - **SETVAL**
  - /\* arg for semctl system calls. \*/  
union semun { int val; /\* value for SETVAL \*/  
struct semid\_ds \*buf; /\* buffer for IPC\_STAT & IPC\_SET \*/  
ushort \*array; /\* array for GETALL & SETALL \*/  
struct seminfo \*\_\_buf; /\* buffer for IPC\_INFO \*/  
void \*\_\_pad; };



## Operações sobre semáforos

- **int semop(int semid, struct sembuf \*sops, unsigned nsops);**
  - Semid= identificador retornado pela operação semget
  - struct sembuf {
    - ushort sem\_num; /\* semaphore index in array \*/
    - short sem\_op; /\* semaphore operation \*/
    - short sem\_flg; /\* operation flags \*/
  - };
  - sem\_op= -1 (wait), 1 (signal), 0 dorme até o valor do semáforo for 0
  - sem\_flg= IPC\_NOWAIT = falha se o semáforo não está disponível
  - nsops= número de operações



## Exemplo

```
#define SHM_SIZE 4096
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/ipc.h>

union semun{
    int val;    struct semid_ds *buf;    unsigned short *array;    struct seminfo *__buf;
};

int main(){
    int shmid;
    int semid;
    char* saddr;
    union semun arg;
    struct sembuf wait={0, -1, 0};
    struct sembuf signal={0, 1, 0};

    // Allocate
    shmid= shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT | S_IRUSR | S_IWUSR);
    semid= semget(IPC_PRIVATE, 1, IPC_CREAT | S_IRUSR | S_IWUSR);

    saddr= (char *)shmat(shmid, 0, 0);
    // Inicializa
    arg.val= 1;
    semctl(semid, 0, SETVAL, arg);
```



## Exemplo (2)

```
if (fork()==0){//children
    saddr[0]=65;
    printf("Filho pegando semaforo\n");
    if (semop(semid, &wait, 1)==-1){
        perror("semop"); return 1;
    }
    printf("Filho  secou critica\n");
    sleep(20);
    if (semop(semid, &signal, 1) == -1){
        perror("semop");
        return 1;
    }
    printf("Filho liberou semaforo\n");
    return 0;
}
saddr[1]=66;
sleep(2);
printf("Pai esperando semaforo\n");
if (semop(semid, &wait, 1) == -1){
    perror(" semop");
    return 1;
}
printf(" Pai secou critica %s" , saddr);
if (semop(semid, &signal, 1) == -1){
    perror(" semop");
    return 1;
}
printf("Pai liberou semaforo\n");
shmctl(shmid, IPC_RMID, 0);
return 0;
}
return 0;
```



## Problemas clássicos de sincronização

- Problema de buffer de tamanho limitado
- Problema dos leitores e escritores
- Problema dos filósofos comilões



## Buffer de tamanho limitado

- $N$  posições, cada posição armazena um elemento
- Semáforo **mutex** inicializado com 1
- Semáforo **full** inicializado com 0
- Semáforo **empty** inicializado com  $N$ .



## Produtor

```
while (true) {  
  
    // produce an item  
  
    wait (empty);  
    wait (mutex);  
  
    // add the item to the buffer  
  
    signal (mutex);  
    signal (full);  
  
}
```



## Consumidor

```
while (true) {  
    wait (full);  
    wait (mutex);  
  
    // remove an item from buffer  
  
    signal (mutex);  
    signal (empty);  
  
    // consume the removed item  
  
}
```



## Problema dos leitores e escritores

- Um conjunto de dados compartilhado entre processos
  - Leitores –somente realizam a leitura
  - Escritores – podem ler ou escrever
- Problema– permitir múltiplos leitores simultaneamente. Somente um único escritor pode escrever ao mesmo tempo.
- Dados compartilhados
  - Buffer
  - Semáforo **mutex** inicializado com 1.
  - Semáforo **wrt** inicializado com 1.
  - Inteiro **readcount** inicializado com 0. Número de leitores lendo o buffer compartilhado



## Escritor

```
while (true) {  
    wait (wrt) ;  
  
    // writing is performed  
  
    signal (wrt) ;  
}
```



## Leitor

```
while (true) {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readcount == 1) wait (wrt) ;  
    signal (mutex)  
  
    // reading is performed  
  
    wait (mutex) ;  
    readcount -- ;  
    if (readcount == 0) signal (wrt) ;  
    signal (mutex) ;  
}
```



## Leitores e escritores

- Nesta solução: o fato de ter um escritor executando, não impede que outro leitor entre na seção crítica
- Quando um escritor sai da seção crítica `signal(wrt)`, o novo processo a entrar pode ser tanto um leitor, quanto um escritor
- Abandono de processos
- Solução alternativa: escritor tem prioridade
  - Caso um escritor esteja esperando, nenhum leitor pode entrar.

