Sistemas Operacionais

ROTEIRO LABORATÓRIO 4 - Memória Compartilhada (*Shared Memory*)

Conceito: Memória Compartilhada (Shared Memory)

Trata-se de um mecanismo de IPC (*Inter-Process Communication*) que cria uma região de memória que pode ser compartilhada por dois ou mais processos. Após a criação, a região deve ser "ligada" ou anexada (*attached*) ao processo. Ao ser ligada a um processo, a região de memória criada passa a fazer **parte do seu espaço de endereçamento**. Com isso, o processo pode ler ou escrever no segmento, de acordo com as permissões definidas na operação de "attachment".

O S.O. oferece chamadas de sistemas para criar regiões de memória compartilhada, mas **não** se envolve diretamente na comunicação entre os processos. Isto é, as regiões de memória compartilhada e os processos que as utilizam são gerenciados pelo núcleo, **mas o acesso ao conteúdo** (dados armazenados na memória compartilhada) é feito diretamente pelos processos, sem que seja necessária nenhuma chamada de sistema.

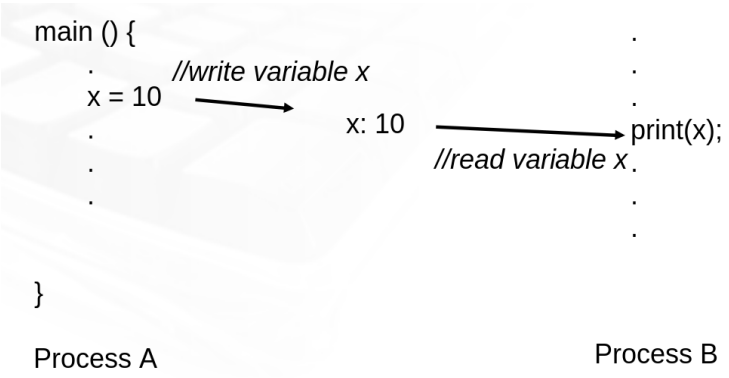

Fig. 1: A variável x se encontra em uma região de memória compartilhada entre os processos A e B. Se A faz alguma modificação na região compartilhada, isso é visto por todos os outros processos que compartilham a região (neste exemplo, o processo B).

As principais vantagens deste tipo de mecanismo de IPC são:

> **Eficiência:** É a maneira mais rápida para dois processos efetuarem uma troca de dados. Os dados não precisam ser passados ao kernel para que este os repasse aos outros processos. Em outras palavras, ao contrário de pipes, uma leitura ou escrita nos dados é feita sem nenhuma chamada de sistema. O acesso à memória é direto.
>
> **Acesso randômico:** Diferentemente dos *pipes*, é possível acessar uma parte específica de uma estrutura de dados que está sendo comunicada (por exemplo, uma posição de um vetor ou um campo de uma struct).

… Mas há uma desvantagem importante: **NÃO existe nenhum mecanismo automático (implícito) de sincronização,** podendo exigir, o uso de mecanismos externos (programados pelo programador) para controlar ou inibir inconsistências. Por exemplo, imaginem que dois processos tentem escrever "ao mesmo tempo" na região de memória compartilhada. Ou ainda, um processo tente ler um dado dessa região mas nenhum processo escreveu o dado ainda. Vejam que no caso dos *pipes*, esse mecanismo implícito de sincronização existe. Por exemplo, quando um processo tenta ler de um *pipe* que não contém nenhum dado, o processo fica bloqueado até que um segundo

processo escreva nesse mesmo *pipe*. Ou ainda, se um processo tenta escrever em um *pipe* que já esteja "cheio", ele também vai ficar bloqueado.

Criação e uso de uma área de memória compartilhada no UNIX

A criação e uso de um segmento de memória compartilhada no UNIX se faz por uma sequência de passos:

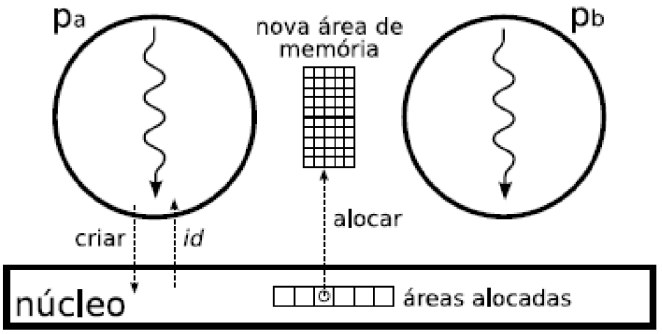

Fig. 2: Passo 1 – Criação de um segmento de memória "compartilhável"

Passo 1) O processo p_a solicita ao núcleo (kernel) a criação de uma área de memória compartilhada, informando o tamanho e as permissões de acesso; o retorno dessa operação é um identificador (*id*) da área criada.

Para praticar o "Passo 1" vamos começar usando a seguinte função (que é um *wrapper* de uma chamada de sistema):

`shmget()` é a função usada para criar uma área de memória compartilhada de tamanho **`size`**.

```
shmid = shmget( key_type key,  /* chave de identificação
                    int size,/* tamanho do segmento */
                int shmflag)      /* flags de permissão */
```

Essa função é encarregada de buscar o elemento especificado pela chave de acesso **`key`**, caso esse elemento não exista, pode criar um novo segmento de memória compartilhada OU retornar erro (dependendo do que for especificado no campo **`shmflag`**).

Em caso de sucesso, a função devolve o identificador do segmento de memória compartilhada, caso contrário retorna -1.

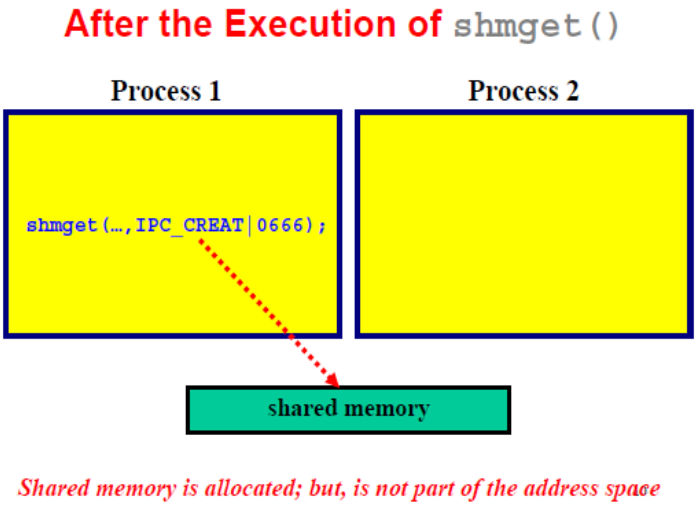

Fig. 3: Criação de um segmento de memória via ***shmget(…,IPC_CREAT ...)****. Após a criação, o segmento* ***ainda não está acessível*** *pois ele não foi "anexado" ao espaço de endereçamento do processo*

```
// test_shmget.c()
// Criação de um novo segmento de código que pode ser
// compartilhado entre processos
#include <errno.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define ADDKEY 123
// OBS: SHM_R=0400 SHM_W=200 SHM_R AND SHM_W = 0600

int main() {
int shmid ; /* identificador da memória comum */
int size = 1024 ;
char *path="./" ;
if (( shmid = shmget(ftok(path,ADDKEY), size,
                     IPC_CREAT| IPC_EXCL|SHM_R|SHM_W)) == -1) {
      perror("Erro no shmget") ;
      exit(1) ;
}
printf("Identificador do segmento: %d \n",shmid) ;
printf("Este segmento e associado a chave unica: %d\n",
ftok(path,ADDKEY)) ;
exit(0);
}
```

TAREFAS:

1) Compile e rode o programa **`test_shmget.c`**. ATENÇÃO! EXECUTE APENAS UMA VEZ!!! Após executar, rode no shell o comando "**`ipcs -m`**" e procure na coluna "**`shmid`**" o identificador do segmento).

IPC - Inter-Process Communication

No decorrer de sua existência, diferentes extensões foram desenvolvidas no UNIX para viabilizar a comunicação efetiva entre processos. Tais extensões são comumente conhecidas pela sigla IPC (InterProcess Communication). Além de Pipes (visto no

último roteiro) existem outros mecanismos de IPC implementados no UNIX (ex: memória compartilhada, semáforos e filas de mensagens)... A maior parte dos sistemas UNIX fornece ao usuário um conjunto de comandos que permitem o acesso às informações relacionadas aos mecanismos implementados em IPC. Os comandos **`ipcs`** e **`ipcrm`** são bastante úteis ao programador durante o desenvolvimento de aplicações. O comando **`ipcs -<recurso>`** fornece informações atualizadas de cada um dos recursos IPC implementados no sistema. Em particular, `ipcs -m` lista as informações relativas aos segmentos de memória compartilhada.

2) Rode o programa **`test_shmget.c`** uma segunda vez. O que acontece?

Na TAREFA 2 você verificou que o programa sinaliza um erro. Para entender isso melhor, precisamos primeiramente detalhar os parâmetros da função **`shmget.`**

`key`: Esta "chave de acesso" define um identificador único no sistema para a área de memória que se quer criar ou à qual se quer ligar. Uma chave nada mais é do que um valor inteiro longo. Em geral, chaves de acesso são utilizadas para identificar estruturas de dados que serão referenciadas por programas. No caso dessa estrutura de dados ser uma região de memória compartilhada, todos os processos que quiserem se conectar a essa região devem referenciar a mesma chave de acesso **`key`** utilizada durante a criação dessa região. A chave é do tipo `long`, então qualquer número poderia ser usado como chave. Mas para evitar que dois processos "não relacionados" corram o risco de usar a mesma chave (por coincidência!), o mais indicado é criar chaves por meio da função **`ftok().`**

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(char *path, int proj)
```

Valor de retorno: valor de uma chave única para todo o sistema ou -1 em caso de erro.

A função ftok() usa o nome do arquivo apontado por **path**, que é único no sistema, como uma cadeia de caracteres, e o combina com um identificador **proj** para gerar uma chave do tipo key_t no sistema IPC.

`size`: é o tamanho em bytes do segmento de memória compartilhada.

`shmflag`: especifica as permissões do segmento por meio de um OR bit-a-bit, os seguintes valores podem ser combinados:

- **`IPC_CREAT`**: cria o segmento, caso ele já não exista;
- **`IPC_EXCL`**: caso se queira exclusivamente criar o segmento (ou seja, se o segmento já existir, a função retornará -1)
- **`0---`**: flags de permissão de acesso rwx para usuário-grupo-outros (ex:**`0664`**)

  Também podem ser usadas constantes pré-definidas... ex:

  `SHM_R` (~**`0400`**); **`SHM_W`** (~**`0200`**)

Ex de parâmetro `shmflag`: `shmget(..., ..., IPC_CREAT|IPC_EXCL|0640)`

Se **`shmflg`** for `0` (zero), a função não cria nenhum segmento novo. Ela apenas retorna o ***id*** de um segmento já existente. Um processo deve portanto utilizar **`shmget(..., ..., 0)`** para receber o **id** de um segmento já criado por outro processo, e fazer em seguida um "attachment" desse segmento… mas controlem-se… nós só faremos isso nos Passos 2 e 3.

Voltando à TAREFA 2, observem a chamada **`shmget`** usada no test_shmget.c:

```
shmget(ftok(path,ADDKEY), size, IPC_CREAT|IPC_EXCL|SHM_R|SHM_W)) == -1)
```

Vamos por partes:

`ftok(path,ADDKEY)`: Aqui estamos criando uma chave de acesso (tipo **`key_t`**), utilizando o *patch* do diretório corrente e o combinando com o valor 123.

`IPC_EXCL`: aqui é fácil… estamos dizendo ao SO que queremos criar um segmento NOVO. Se já existir um segmento com o mesmo identificador gerado por **`ftok(path,ADDKEY)`**, o SO deve retornar -1.

Agora vocês conseguem entender por que houve o erro na TAREFA 2? É que na primeira vez que vocês rodaram `test_shmget.c`, foi criado um segmento de memória cuja chave de acesso é o valor gerado por `ftok(path,ADDKEY)`. Na segunda vez que vocês rodaram o mesmo código, o segmento já existia.

PERCEBAM… o segmento foi criado e alocado na RAM pelo Sistema Operacional… Quando o processo "criador" morre, **o segmento continua existindo!** Isso vocês observaram usando o comando `ipcs -m` … Reparem também que o segmento ainda não foi anexado a nenhum processo (observem a coluna `nattch` após o comando `ipcs -m`) … Acalmem-se! Faremos isso!

Continuando a parte prática…

Quando um novo segmento de memória é criado, é criada uma estrutura `shmid_ds`:

```
 struct shmid_ds {
  struct ipc_perm shm_perm;   /* operation permissions */
  int shm_segsz;         /* size of segment (bytes) */
  time_t shm_atime;      /* last attach time */
  time_t shm_dtime;      /* last detach time */
  time_t shm_ctime;      /* last change time */
  unsigned short shm_cpid;    /* pid of creator */
  unsigned short shm_lpid;    /* pid of last operator */
  short shm_nattch;      /* no. of current attaches */
};
```

Nela são mantidas as informações sobre o segmento (ex: permissões de acesso definidas pelo parâmetro `shmflg` da chamada `shmget()` em que o segmento foi criado).

Já os campos no membro `shm_perm` são os seguintes:

```
struct ipc_perm {
      key_t key;
      ushort uid;    /* owner euid and egid (e from effective) */
      ushort gid;
      ushort cuid;       /* creator euid and egid */
      ushort cgid;
```

```
        ushort mode;       /* lower 9 bits of shmflg */
        ushort seq;    /* sequence number */
    };
```

A chamada de sistema `shmctl()` permite examinar ou modificar as informações relativas ao segmento de memória compartilhada: ela permite ao usuário receber informações relacionadas ao segmento, definir o proprietário ou grupo, especificar permissões de acesso e, adicionalmente, destruir o segmento.

```
#include <sys/ipc.h>
#include <sys/shm.h>
   int shmctl( int shmid,
               int cmd,
               struct shmid_ds *buf);
```

TAREFA:

3) Compile e rode o programa `test_shmctl.c` (lembre-se de alterar a variável `path` para o mesmo usado na TAREFA 1!!!!!!!!!!!). Após executar, rode na bash o comando "ipcs -m". Você ainda vê o segmento que foi criado na lista? O que aconteceu com ele?

```
/* arquivo test_shmctl.c */

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
//...

#define ADDKEY 123
struct shmid_ds buf ;

int main() {
     char *path="nome_de_um_arq_ou_dir_existente" ; //TROQUE O VALOR!
     int shmid ;
     int size = 1024 ;
     /* recuperação do identificador do segmento associado à chave 123 */
     if (( shmid = shmget(ftok(path,(key_t)ADDKEY),size,0)) == -1 ) {
          perror ("Erro shmget()") ;
          exit(1) ; }

     /* recuperação das informações relativas ao segmento */
     if ( shmctl(shmid,IPC_STAT,&buf) == -1){
          perror("Erro shmctl()") ;
          exit(1) ;}
     printf("ESTADO DO SEGMENTO DE MEMORIA COMPARTILHADA %d\n",shmid) ;
     printf("ID do usuario proprietario: %d\n",buf.shm_perm.uid) ;
     printf("ID do grupo do proprietario: %d\n",buf.shm_perm.gid) ;
     printf("ID do usuario criador: %d\n",buf.shm_perm.cuid) ;
     printf("ID do grupo criador: %d\n",buf.shm_perm.cgid) ;
     printf("Modo de acesso: %d\n",buf.shm_perm.mode) ;
     printf("Tamanho da zona de memoria: %ld\n",buf.shm_segsz) ;
     printf("pid do criador: %d\n",buf.shm_cpid) ;
     printf("pid (ultima operacao): %d\n",buf.shm_lpid) ;

     /* destruicao do segmento */
     if ((shmctl(shmid, IPC_RMID, NULL)) == -1){
```

```
            perror("Erro shmctl()") ;
            exit(1) ; }
        exit(0);
}
```

No código do programa, o processo tem acesso às informações sobre o segmento de memória por meio da chamada `shmctl(shmid,IPC_STAT,&buf):`

- No primeiro parâmetro ele passa o identificador do segmento;
- no segundo parâmetro é passada a flag `IPC_STAT`, que indica justamente que a chamada `shmctl` vai ser usada para recuperar as informações sobre o segmento;
- o terceiro parâmetro é onde será armazenada a struct `shmid_ds`.

Na parte final do código (em azul) é feita uma segunda chamada a `shmctl`, mas com outros parâmetros: **`shmctl(shmid, IPC_RMID, NULL)`**. Aqui o 2o. parâmetro contém a flag **`IPC_RMID`** (equivale a 0), o que significa que o segmento de memória será destruído. Mas para que a chamada seja executada com sucesso, o usuário deve ser **o proprietário**, o **criador**, ou o **super-usuário** para realizar esta operação. Os outros valores possíveis para o 2o. parâmetro (**`cmd`**) são:

> **`IPC_SET`** (1): é usada para alterar informações sobre a memória compartilhada. Os novos valores são copiados da estrutura apontada por **`buf`**. A hora da modificação é também atualizada.
>
> **`IPC_STAT`** (2): é usada para copiar a informação sobre a memória compartilhada. Copia para a estrutura apontada por **`buf`**;

Obs.: O super usuário pode ainda evitar ou permitir o swap do segmento compartilhado usando os valores **`SHM_LOCK`** (3), para evitar o swap, e **`SHM_UNLOCK`** (4), para permitir o swap.

IMPORTANTE: Quando **`shmctl(shmid, IPC_RMID, NULL)`** é usado, o segmento só é removido quando **o último processo que está** ligado (*attached*) a ele é finalmente desligado dele. Além disso, é uma boa prática que cada segmento compartilhado deva ser explicitamente desalocado (removido) usando **`shmctl`** após o seu uso para evitar problemas de limite máximo no número de segmentos compartilhados alocados no sistema. A invocação de **`exit()`** e **`exec() por`** por um processo que tenha anexado um segmento de memória a seu espaço de endereçamento não extingue esse espaço de endereçamento.

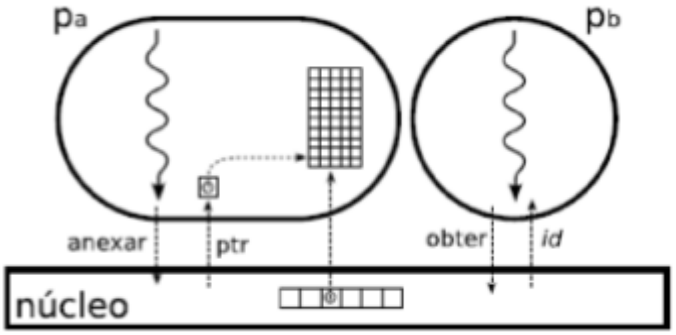

Fig. 4: Passos 2 e 3 – Attachment de um segmento de memória ao processo p_a . Como resultado o processo obtém um ponteiro para esse segmento.

Passo 2) O processo p_a, após ter criado um segmento de memória "compartilhável", solicita ao núcleo que a área recém-criada seja anexada ao seu espaço de endereçamento. Essa operação retorna para p_a um ponteiro para a nova área de memória (*ptr*), que pode então ser acessada pelo processo p_a.

Passo 3) O processo p_b obtém o identificador (*id*) da área de memória criada por p_a.

Bom, finalmente, agora que vocês já sabem criar, verificar e destruir um segmento de memória "compartilhável", vamos ver como anexá-lo ao espaço de endereçamento de um processo. Para isso, no Passo 2, temos que primeiramente obter o id do segmento… isso vocês já sabem como obter: usando **`shmget`**, seja ao criar um novo segmento (a função retorna justamente o id desse segmento criado) seja passando a chave de um segmento criado anteriormente.

Mas para executar o Passo 2 também precisamos de uma outra chamada de sistema:

```
void *shmat(int   shm_id,  /*ID do segmento obtido via shmget() */
            void *shm_ptr /* Endereço do acoplamento do segmento */
            int   flag);  /* Igual a SHM_RONLY, caso só leitura,
                             ou 0 (zero), caso contrário */
```

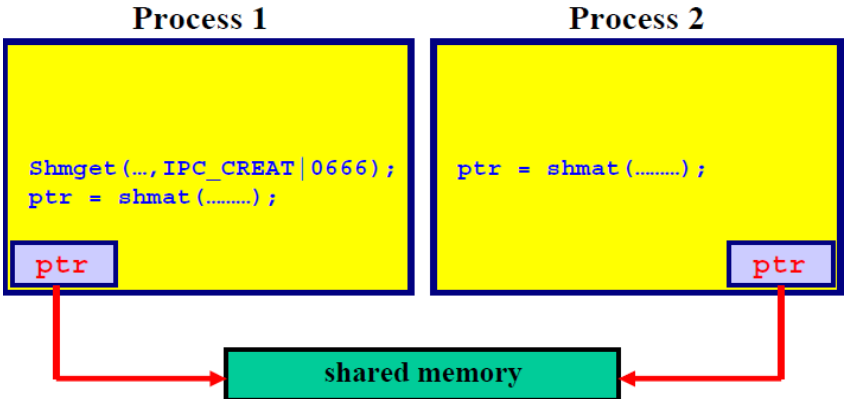

Fig. 5: Ligação à Memória Compartilhada usando shmat()

O processo usa a função **`shmat()`** para se ligar a um segmento de memória existente referenciado pelo identificador **`shm_id`**. A função retorna um **ponteiro** para a memória alocada e esta torna-se parte do espaço de endereçamento do processo. Os outros dois argumentos são:

- **`shm_ptr`**: É um ponteiro que especifica onde, no espaço de endereçamento do processo, se quer mapear (acoplar) a memória compartilhada. Ainda não estudamos memória virtual, mas para resumir a estória, o programador poderia escolher (dentre os endereços de memória não alocados… aqueles que estão na "faixa" de HEAP) um

endereço para este segmento de memória compartilhado. Se for especificado 0 (NULL), que é o mais o usual, o SO escolhe ele mesmo um endereço disponível (dentro da faixa de HEAP) para acoplar o segmento no espaço de endereços do processo.

- **`flags`**:
  - Se igual a **`SHM_RND`**: indica ao sistema que o endereço especificado no segundo argumento deve ser arredondado (p/ baixo) para um múltiplo do tamanho da página (aqui, novamente, vocês só vão entender quando forem estudar memória virtual) .
  - Se igual a **`SHM_RDONLY`**: indica que o segmento será *read only*.
  - Se igual a **`0`** (zero): indica leitura e escrita.

Observe este exemplo:

```
char *p ;
...
p = (int *) shmat(shmid, 0, 0);
```

… aqui um processo está anexando ao seu espaço de endereçamento um segmento de memória identificado por `shmid` em modo `RW` (*read-write*). `p` recebe um ponteiro para o primeiro byte desse segmento. Desta forma consegue-se ter um ponteiro para todo o segmento de memória (basta ir deslocando `p`, assim como fazemos com vetores).

Agora vejam este segundo exemplo:

```
typedef struct
{
  int x;
  int y;
} Coord;
Coord *mycoords;
...
int size = 10*sizeof(Coord);
shmid = shmget(some_key, size, IPC_CREAT|0600 )
...
mycoords = (Coord*) shmat(shmid, 0, 0);
mycoords[2].x=5; //escrita na mem. compartilhada
mycoords[2].y=10;
```

… aqui é possível acessar a memória de forma estruturada, por meio de uma struct.

TAREFA:

4) Rode o programa **`test_shmget.c`** (para criar o segmento de memória novamente… lembrando que na Tarefa 3 você destruiu esse segmento). Agora compile e rode (rode apenas UMA VEZ!) o programa **`test_shmat.c`** **em background, usando '&'** no final (lembre-se de alterar a variável **`path`** para o mesmo valor usado na TAREFA 1). Após executar `test_shmat.c`, rode na shell o comando "**`ipcs -m`**" e observe a coluna "**`nattch`**". Que valor ela mostra para a linha correspondente ao segmento de memória que foi criado pelo **`test_shmget.c`**? [Responda no formulário online]

Passo 4) O processo p_b solicita ao núcleo que a área de memória seja anexada ao seu espaço de endereçamento e recebe um ponteiro (*ptr*) para o acesso à mesma.

Passo 5) Os processos p_a e p_b acessam a área de memória compartilhada por meio dos ponteiros informados pelo núcleo.

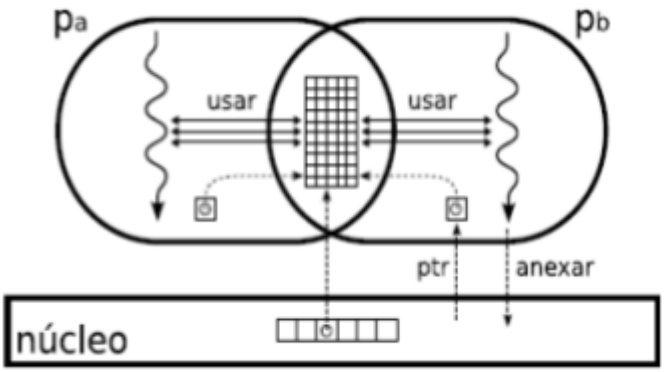

Fig. 6: Passos 5 e 6 – Attachment do segmento de memória ao processo p_b

Para executar os passos 4 e 5, basta que o processo p_b obtenha o id do seguimento de memória (usando **`shmget()`**) e depois faça o "attachment" (usando **`shmat`**).

TAREFAS:

5) Compile e rode o programa **`test_shmat2.c`** **em background** (lembre-se de alterar a variável **`path`** para o mesmo valor usado na TAREFA 1). Após executar, antes que os processos morram, rode no shell o comando "**`ipcs -m`**" e observe a coluna "**`nattch`**". Que valor ela mostra para a linha correspondente ao segmento de memória que foi criado pelo **`test_shmget.c`**? [Responda no formulário online]

6) Agora, verifique se **`test_shmat`** e **`test_shmat2`** já terminaram (comando **`ps`** no mesmo terminal onde foram rodados os programas). Se não terminaram, termine eles usando **`kill`**. Agora rode novamente **`test_shmat`** e, antes que ele termine, rode **`test_shmctl`** (lembra? esse programa deveria remover o segmento de memória criado por **`test_shmget`**). Usando "**`ipcs -m`**" você consegue ver que o segmento ainda está no sistema! Mas observe na coluna "chave" para esse segmento…. qual é o valor mostrado? [Responda no formulário online]

7) Por fim, tente rodar **`test_shmat2.c`** novamente. Ele consegue anexar o segmento ao seu espaço de endereçamento? [Responda no formulário online]

Na TAREFA 6, o que aconteceu é que o SO não permite excluir um segmento enquanto ele estiver "atachado" a um processo. Com isso, quando um processo tenta destruir um segmento de memória, o SO, ao invés de destruí-lo, torna-o "PRIVATE" (vejam que ao fazer um "`ipcs -m`" a coluna "`chave`" tem seu valor alterado).

Segmentos de memória do tipo PRIVATE

Uma forma muito comum de se criar um segmento de memória "compartilhável" é NÃO associar nenhuma chave a ele, criando assim um segmento "PRIVATE":

```
shmid = shmget (IPC_PRIVATE , … , …)
```

O processo criador recebe como retorno da chamada `shmget()` o *id* desse segmento e com esse id é possível fazer o *attachment*. Observem que, como **não há chave associada**. Para que outros processos possam acessar esse mesmo segmento eles devem conhecer o seu *id*. Mas como? Pensem… quando o processo criador do segmento fizer `fork()`, seus descendentes poderão acessar o segmento uma vez que eles "herdaram" essa informação armazenada em alguma variável (copiada do processo pai). Com isso, geralmente, apenas processos relacionados (ex: que têm "ancestrais em comum") usam segmentos PRIVATE. A grande vantagem desse tipo de segmento é que torna-se **impossível** um processo não relacionado com o processo criador, por coincidência (ou maldade mesmo!), usar uma mesma chave e acabar acessando um segmento de memória compartilhado que ele não deveria.

Voltando à TAREFA 6, uma vez que o SO transformou o segmento em PRIVATE, outros processos que tentarem anexar esse mesmo segmento não o conseguirão. Quando um segmento torna-se PRIVATE, não é mais possível obter o id desse segmento por meio da chamada `shmget()`.

TAREFA:

8) Agora altere o código de **`test_shmat`** e **`test_shmat2`** de forma que ambos os processos, após realizarem o attachment, imprimam o endereço de memória retornado pelo **`shmat()`**. Os endereços do ponteiro **`mem`** de cada processo são iguais ou diferentes? [Responda no formulário online]

Para entender o que aconteceu na TAREFA 8 vocês têm que usar o conceito de Memória Virtual. Na verdade, esses endereços retornados por **`shmat()`** em cada processo correspondem a endereços lógicos (ou endereços virtuais), que serão convertidos em um MESMO endereço físico! Veremos isso melhor dentro de algumas aulas.

TAREFA:

9) Agora vamos zerar tudo… termine os processos **`test_shmat.c`** e **`test_shmat2.c`** e rode novamente **`test_shmctl`** para excluir o segmento. Então rode **`test_shmget.c`** para criar o segmento mais uma vez. Agora nós vamos trocar a ordem… você vai executar

`test_shmat2` ANTES de executar `test_shmat`. O que aconteceu nesse caso? [Responda no formulário online]

[EXTRA]

Os programas "normais" não precisam se preocupar com endereços físicos enquanto são executados em máquinas que usam memória virtual… isso porque é muito mais conveniente trabalhar com um espaço de endereço virtual, tendo todas as suas conveniências. Além disso, nem todos os endereços virtuais têm um endereço físico necessariamente, eles podem, por exemplo, pertencer a arquivos mapeados (área de SWAP). No entanto, às vezes, pode ser interessante ver esse mapeamento, mesmo sendo um usuário comum.

Para este propósito, o kernel Linux expõe seu mapeamento para os usuários por meio de um conjunto de arquivos no diretório /proc . A documentação pode ser encontrada em:

https://www.mjmwired.net/kernel/Documentation/vm/pagemap.txt

Pequeno resumo:

1. `/proc/$pid/maps` *provides a list of mappings of virtual addresses together with additional information, such as the corresponding file for mapped files.*
2. `/proc/$pid/pagemap` *provides more information about each mapped page, including the physical address if it exists.*

   Obs: ***sudo*** *is required to read /proc/[PID]/pagemap*